FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO
Departamento de Engenharia Electrotécnica e de Computadores

Transparent Metropolitan Vehicular Network — Design and Fast Prototyping Methodology

Gustavo João Alves Marques Carneiro

Dissertação submetida para satisfação parcial dos
requisitos do grau de doutor
em
Engenharia Electrotécnica e de Computadores

Dissertação realizada sob a supervisão do
Professor Doutor Manuel Alberto Pereira Ricardo,
do Departamento de Engenharia Electrotécnica e de Computadores
da Faculdade de Engenharia da Universidade do Porto

Porto, Julho de 2012

Abstract

Future public transportation systems will provide broadband access to passengers carrying legacy terminals with 802.11 connectivity. Passengers will be able to communicate with the Internet and with each other, while connected to 802.11 Access Points deployed in vehicles and bus stops / metro stations, and without requiring special mobility or routing protocols to run in their terminals. Existing solutions, such as 802.11s and OLSR, are not efficient and do not scale to large networks, thereby requiring the network to be segmented in many small areas, causing the terminals to change IP address when moving between areas. In this thesis we propose WiMetroNet, a large mesh network of mobile routers (Rbridges) operating at layer 2.5 over heterogeneous wireless technologies that is transparent to end terminals, since they believe they are connected to a simple local area network. This architecture contains an efficient data plane that optimizes the transport of DHCP and ARP traffic, and provides a transparent terminal mobility solution using techniques that minimize the routing overhead for large networks. We offer two techniques to reduce routing overhead associated with terminal mobility, one based on proactively flooding a TTL-limited routing message, and another based on reactively sending back a "binding update" message to the correspondent node when a packet arrives at the old point of attachment. Simulation and analytical results are presented, and the routing protocol is shown to scale to large networks with good data plane results, namely packet delivery rate, delay, and handover interruption time.

During WiMetroNet development, a problem was faced that is recurring in the networking research and development field: the duplication of effort to write first simulation and then implementation code. We posit an alternative development process that takes advantage of the network emulation features of Network Simulator 3 (ns-3) and allows developers to share most code between simulation and implementation of a protocol. Tests show that ns-3 can handle a data plane processing large packets, but has difficulties with small packets. When using ns-3 for implementing the control plane of a protocol, we found that ns-3 can even outperform a dedicated implementation. We further enhance ns-3 with scripting language bindings, a new scalable and simple to use flow monitoring module, and visualization capabilities. As a result, the ns-3 based development of a network protocol and accompanying prototype becomes an attractive alternative to traditional protocol development process, with increased development efficiency.

Resumo

Os sistemas de transportes públicos futuros irão oferecer acesso de banda larga a passageiros que transportam terminais com conectividade 802.11. Os passageiros vão poder comunicar com a Internet e com outros passageiros, estando ligados a pontos de acesso 802.11 instalados em veículos e paragens de autocarro/metro, e sem precisar de correr qualquer protocolo de encaminhamento ou mobilidade nos terminais. As soluções existentes, como por exemplo 802.11s e OLSR, não são eficientes e não escalam para redes grandes, e portanto requerem que a rede seja segmentada em muitas pequenas áreas, fazendo com que os terminais tenham que mudar de endereço IP sempre que mudam de área. Nesta tese propomos WiMetroNet, uma grande rede mesh de routers móveis (Rbridges) que operam na camada 2,5 sobre redes sem fios heterogéneas. Esta rede é transparente para os terminais, pois estes acreditam que estão ligados a uma simples rede local. Esta arquitectura contém um plano de dados eficiente que optimiza o transporte de tráfego DHCP e ARP, e que suporta uma solução de mobilidade transparente de terminais usando técnicas que minimizam o tráfego de controlo para redes grandes. São oferecidas duas técnicas de redução do tráfego de controlo associado à mobilidade dos terminais, uma baseada em proactivamente disseminar uma mensagem de controlo com TTL limitado, e outra baseada no envio reactivo de uma mensagem "binding update" para o nó correspondente sempre que chega um pacote para o antigo ponto de ligação do nó. São apresentados resultados de simulação e analíticos que mostram que o protocolo de encaminhamento escala para redes grandes com bons resultados no plano de dados, nomeadamente taxa de entrega de pacotes e tempo de interrupção durante o *handover*.

Durante o desenvolvimento do WiMetroNet, encontrámos um problema que é recorrente na investigação e desenvolvimento da área de redes: existe duplicação de esforço quando programamos primeiro um simulador e depois uma implementação. Propõe-se nesta tese um processo de desenvolvimento alternativo que tira partido das funcionalidades de emulação que existem no Network Simulator 3 (ns-3) e permite que investigadores partilhem a maior parte do código entre simulação e implementação de um protocolo. Os testes demonstram que o ns-3 consegue suportar um plano de dados que processa pacotes grandes, mas tem dificuldades com pacotes pequenos. Quando usamos o ns-3 para implementar um protocolo no plano de controlo, descobrimos que o ns-3 consegue ter um desempenho melhor do que uma implementação dedicada. Para tornar o desenvolvimento ainda mais rápido,

melhorámos o ns-3 com suporte para uma linguagem de *scripting* de alto nível, um novo módulo simples e escalável para monitorização de fluxos, e funcionalidades de visualização com foco na depuração. Como resultado, o desenvolvimento de um protocolo de redes e respectivo protótipo baseado em ns-3 torna-se uma atractiva alternativa ao processo tradicional de desenvolvimento de protocolos, com ganhos no tempo de desenvolvimento.

Acknowledgments

The author would like thank a number of people for making this work possible. First and foremost, prof. Manuel Ricardo for always believing in me and for invaluable advice and help. No doubt his experience and commitment had a big impact on the resulting work presented here.

A big thanks to all my colleagues at INESC TEC. Two in particular deserve a special mention. Pedro Fortuna, for his contribution as co-author of the base WiMetroNet architecture and design. Jaime Dias for always giving great feedback and coming up with good ideas. I wish you both the best of luck on your theses. Thanks to all my other colleagues for their support and friendship; some of the colleagues that supported me are (in alphabetical order): António Pinto, Carlos Pinho, Filipe Sousa, Filipe Teixeira, Helder Fontes, Hermes del Monego, Jaime Dias, Nuno Salta, Pedro Fortuna, Renata Rodrigues, Ricardo Duarte, Rui Campos, Saravanan Kandasamy, Tânia Calçada. Apologies to anyone I forget.

The author would like to thank the institutions that supported this work. This work was funded by the Portuguese government through Fundação para a Ciência e Tecnologia (FCT) grant SFRH/BD/23456/2005. No doubt FCT is the main driver of higher education in Portugal. INESC TEC, for providing additional funding and a rich environment for research. The Wireless Networks group at INESC TEC, UTM, is clearly a great environment to do research, with a network of support where no PhD student will have to feel alone. The Faculty of Engineering of University of Porto (FEUP), for accepting me as their student and providing excellent education.

The ns-3 team members, especially Tom Henderson, Mathieu Lacage, and Craig Dowell, among others, are also deserving of praise for their role in creating the modern and powerful simulator that is Network Simulator 3 (ns-3), which is used extensively in this thesis.

Finally, I would like to thank my family, my girlfriend Claudia, and friends, for their encouragement and moral support.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Wireless network for metropolitan transports

In this thesis, we address the moving networks research area. Our particular network — the *WiMetroNet* [1] — is a private network possibly owned by a consortium of companies which jointly operate public transportation vehicles such as buses, trams, and taxis. The addressed scenario provides broadband wireless access to a few thousands of vehicles and vehicle stops. We assume each vehicle (e.g. bus, tram, subway train, or taxi) will use its broadband access to operate services such as video-surveillance, video broadcast, and video/voice calls. Besides, each vehicle is expected to provide wireless access to its passengers, which carry portable and conventional equipments with standard IEEE 802.11 (WLAN) interfaces and a bare IP communications stack. Passengers may access the Internet not only from the vehicles but also from the stops while waiting for the vehicles, and are allowed to communicate between themselves; they can, for instance, exchange files, play games or establish voice and video communications using their applications.

Vehicles get a broadband wireless access by using heterogeneous wireless technologies, namely IEEE 802.16 (WMAN, WiMax), IEEE 802.11, and 3GPP Universal Mobile Telecommunications System (UMTS). Each vehicle has a WMAN access which may not be accessible from every place, a WLAN access which is used when the vehicle approaches some stops, and an UMTS access which it uses when uncovered by the other technologies. Vehicles and vehicle stops are equipped with a communication equipment — the *Rbridge* — which manages the wireless broadband access and serves one or more WLAN Access Point (AP) located inside the vehicle, to which the passenger or other vehicle equipments can associate. Fig. 1.1 presents the WiMetroNet

reference architecture. When, for instance, a passenger arrives to a tram station, he gets a secure wireless access and IP connectivity. While moving from the tram station to a tram, from the tram to the arrival station, and from there to a bus, the passenger is expected to maintain its connection and observe no considerable degradation on the quality of his communications.

The WiMetroNet is a mesh network of moving Rbridges. It is auto-configurable, and it operates at layer 2.5 over heterogeneous wireless technologies. A new routing protocol is proposed, and secure mechanisms for authorization, authentication and confidentiality are used. A passenger's equipment will see WiMetroNet as its LAN, thus being one IP hop away from the other terminals attached to WiMetroNet and from its default router. We say that the network is "transparent" from the point of view of end terminals: just like a transparent window glass allows photons to cross its structure with very little interference, so does WiMetroNet allow a terminal's packets to cross its mesh network structure with little interference. Because terminals are always virtually on the same LAN, while roaming they maintain their IP addresses. Moreover, depending on the mobile terminal implementation, in some cases DHCP renew may not even be necessary while roaming, only when the lease expires. We will refer to the class of networks similar to WiMetroNet as *Wireless Network for Metropolitan Transports* (WNMT).

In order to take advantage of the ability to keep stable IP addresses in mobile terminals, the WiMetroNet network should be reasonably large. We envision many hundreds to a few thousands of mesh routers, some mobile, some fixed, and thousands or tens of thousands of mobile terminals. A traditional IEEE 802 based layer 2 forwarding architecture, which we try to emulate, is not suitable due to the way broadcasts are handled [2]. For instance, if each terminal sends one broadcast ARP packet per minute, for a network with ten thousand terminals this translates into $\frac{10000-1}{60} = 167$ packets per second received by every other host, leading to a virtual collapse of the network. Not only does a simple 802.1D bridged solution exhibit this problem, but also the new 802.11s wireless mesh standard. In the control plane, similar scalability problems exist. Adhoc routing protocols rely on the ability to flood the network to discover routes. The flooding can be periodic, in the case of proactive routing protocols like Optimized Link State Routing (OLSR) [3], or reactive as in Adhoc On-demand Distance Vector (AODV) [4], but both cases result in a considerable fraction of the network capacity being consumed just for routing messages. Recent work on VANET routing protocols has focused on reactive adhoc routing protocols augmented with location information in order to provide better scalability. However, the position of a destination node can only be obtained by a sending node by asking

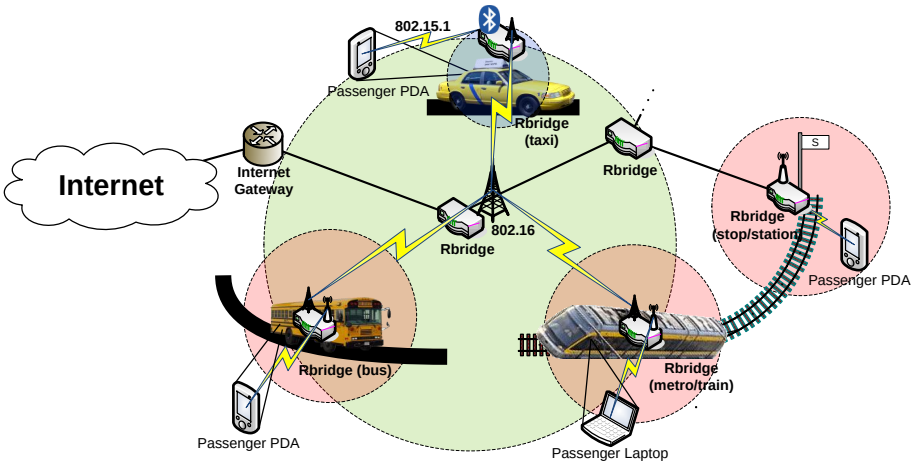


Figure 1.1: The WiMetroNet reference network

a location server, with poor scalability as a result, or by position estimation based on a previous position, with inherent computational complexity and statistical error.

1.2 Protocol research and development

During the course of the WiMetroNet research, we have selected the Network Simulator 3 (ns-3) tool for simulation purposes, mainly due to the need to simulate large networks (with many hundreds to a few thousands of nodes). Since ns-3 is very modular, fast[1], and has a low memory footprint, it was found an ideal tool for research. However, from the experience with using ns-3 for WiMetroNet research and development, we have been able to identify some areas to be improved, as well as new ways to take advantage of ns-3. In this thesis, we propose a new and improved protocol development process which explores the possibility of using ns-3 as a software framework for developing new protocols that can be first simulated and then deployed in a real communications device with only minimal changes. We use the ns-3 emulation benchmark results to guide potential adopters of the proposed development process. The presented development process also includes improvements in the areas of visualization, data output, and scripting abilities. The results of this new process are being applied in the WiMetroNet research, as well as in its spin-off research project called

[1]Fast in essence, although the WiFi model is relatively slow.

"SITMe".

1.3 Thesis objectives

1. *To develop a network architecture for WNMTs.* The network architecture should allow terminals to believe they are connected to a single virtual LAN, support L2 based mobility, and be designed for scalability, while supporting heterogeneous link layer technologies;

2. *To develop optimizations for the above WNMTs architecture to reduce the overhead of terminal mobility.* The base architecture should be simple and support terminal mobility, but as the network scale increases the overhead introduced by the mobility of terminals becomes problematic. The objective is to develop protocol optimizations that allow the terminals locations to be updated quickly without introducing significant routing protocol overhead;

3. *To reduce the time that is spent by researchers in protocol development, from specification to deployment.* One of the main contributors for the time spent in protocol development is the need to develop both a simulation model and then an implementation of the same protocol. This thesis studies the applicability limits of ns-3 emulation in the real world;

4. *To create new ns-3 modules and tools to speed up simulation model development and debugging.* Prior to the work developed in the context of this thesis, ns-3 was lacking in some areas. On one hand, the previously existing ns-3 data collection facilities are either too crude (trace files) or too time consuming (callback tracing) to be practical for researchers. Additionally, writing a simulation script in the C++ programming language is considerably slower, and the debugging more difficult; a high-level scripting language, such as oTCL that exists in ns-2, would be of great help for developers. Finally, in ns-3 the visualization facilities were non-existent or limited, leading to more difficult debugging.

1.4 Original contributions

The following contributions have been produced during the course of this work:

1. WiMetroNet: a novel network architecture designed for a public transportation system that is scalable and appears to end user terminals as just a large Wireless LAN segment. This includes a new scalable data plane, an associated routing protocol — WMRP — that supports mobility of both Rbridges and mobile terminals and distributes the IP/MAC associations needed for the data plane ARP optimizations, and WMRP extensions designed to optimize the mobility of mobile terminals while maintaining a low routing overhead;

2. A new unified simulation/implementation protocol development methodology that takes advantage of the existing ns-3 network emulation functionality. This contribution comprises an evaluation of the packet processing performance, in terms of achievable throughput, packet loss, and round-trip time, of ns-3 working in emulation mode, compared to a pure kernelspace IPv4 forwarding, and also numerous ns-3 improvements to make developing new protocols easier, such as additional ns-3 classes to improve emulation, new visualization capabilities for ns-3, a new scripting framework for ns-3, based on Python, and a new data collection framework for ns-3.

1.5 Thesis organization

The remainder of this thesis is organized as follows. Chapter 2 introduces some of the most relevant state-of-the-art technologies, both in terms of network architecture for vehicular systems, and network simulation. Next, Chapter 3 describes and evaluates our proposal of a network architecture and protocol, including terminal handover optimizations, for the public transportation network scenario introduced here. In Chapter 4, we explain the process of research and development of network protocols, and propose enhancements to help develop and debug simulations, as well as quickly testing new protocols in prototype real equipment. Finally, Chapter 5 summarizes the main conclusions and discusses possible future work.

Chapter 2

WNMT technologies and simulation tools

In this chapter, some relevant state-of-the-art technologies are introduced, discussed, and compared. They allow the reader to better understand the main choices of the thesis approach, always keeping in focus the main context for contributions introduced in Chapter 1. To recall, we aim to develop a network for metropolitan public transport systems where end user mobile terminals are strictly legacy terminals, supporting only WiFi and IPv4 with DHCP. The developed work will specifically focus on terminal handover support while keeping the network scalable. Thus, the quest for a network solution in this scenario requires knowledge of three main subjects. First we want to find out what link layer technologies are available to be used, and what are each technologies' strengths and weaknesses. Second, we want to glue everything together via packet networking; for this, the most relevant networking solutions are explored. Finally, the most important IP based, and above IP, terminal mobility protocols are succinctly explained.

It should be noted that not all related technologies are included in this chapter. Works that we may classify as "related work", rather than "state-of-the-art"[1] are included in each chapter or section in the rest of the document. Related work can be found listed in Chapter 3 (Sec. 3.7) and Chapter 4 (Sec. 4.4.4 and 4.2.5).

[1]The distinction is now always very easy to make, and is rather subjective.

2.1 Link layer technologies

The "link layer" is the layer in the communications stack that allows two nodes to communicate directly, without using additional nodes as relays. For building a communications network, today there are many link layer technologies that can be used. Each one has its own strengths and weaknesses. The list of technologies described in this section is not meant to be exhaustive, merely listing the main technologies that are plausible to be used in the context of public transport networks. These are Ethernet (IEEE 802.3), WiFi (IEEE 802.11), UMTS (3G cellular network), WiMax (IEEE 802.16), and DSRC / IEEE 802.11p. The link layer is almost always implemented as a hardware component, and therefore cannot easily be modified, only selected for use, possibly with some operational parameters slightly adjusted.

2.1.1 Ethernet (IEEE 802.3)

Of all the link layer technologies, Ethernet [5] is one of the oldest and perhaps the most important. The Ethernet standard initially allowed multiple *stations* (network nodes, in the Ethernet terminology) to communicate with each other in a distributed manner by sending and receiving electrical signals over a single shared cable. The main focus at its inception was its decentralized nature, which is achieved thanks to its Carrier Sense Multiple Access with Collision Detection (CSMA/CD) Medium Access Control (MAC) protocol. The CSMA protocol states that a station that wishes to transmit a packet must first listen to signals in the shared cable, and if a signal is detected it has to deffer until the transmission ends; this way, collisions (two stations transmitting at the same time) are mostly avoided.

Because Ethernet stations transmit to a shared medium, they need a way to be able to distinguish which stations are transmitting each packet, and what station is the intended receiver. For this purpose, each station is assigned a unique identifier. This identifier is 48-bits wide, is hard-coded in in the network card, and is guaranteed to be unique by the manufacturer itself. The information transmitted by each station is "framed" by the source and destination addresses, as well as an error detection code called Frame Check Sequence. The packet information, framed by source and destination MAC identifier plus FCS, is called an *Ethernet frame*.

The Ethernet technology was first developed in 1973, with a 3 Mbit/s bitrate and 1 km range, but since then it has been continually evolving. Nowadays, Ethernet is commonly used at 100 Mbit/s speed, but variants

exist that reach 1 Gbit/s, 10 Gbit/s, or even 100 Gbit/s. The electrical specifications of Ethernet cables also evolved considerably, and "Ethernet switches" have replaced the "shared cable" concept, for greater efficiency and reliability. The Ethernet technology was the main foundation of the so called Local Area Network (LAN), which is precisely the concept of a number of stations, geographically close, communicating with each other digitally.

2.1.2 WiFi (IEEE 802.11)

Given the success of Ethernet for wired LANs, the WiFi (IEEE 802.11) was developed using many of the same principles of Ethernet, but adapted to work with radio waves instead of electrical signals on a cable. This allows stations to communicate without cables, over short distances, a concept that was named "Wireless LAN" (WLAN).

The IEEE 802.11 standard offers coverage of up to 100 m outdoor and 30 m indoor, in unlicensed bands. The 802.11b/g works on the 2.4 GHz ISM band, while 802.11a uses the 5 GHz U-NII band. The maximum data rates are 11 Mbit/s for 802.11b, and 54 Mbit/s for 802.11a/g. New radio techniques are expected to enable higher data rates for these technologies: 802.11n will provide up to 300 Mbit/s, and the standards 802.11ac and 802.11ad, under development, are expected to provide bitrates over 1 Gbit/s [6].

The IEEE 802.11 MAC has two main modes of operation. The simplest mode is the so called *adhoc mode*, in which stations can communicate directly with each other, and there is no concept of *connection*, or *association*, similarly to how Ethernet works. The other mode of operation is called *infrastructure mode*, and it requires that wireless stations first associate to a special node, the *access point* (AP), and then they can communicate by transmitting frames to this access point, which retransmits each received frame for the destination station to hear. The infrastructure mode allows two stations to communicate over an extended range, and is better suited to the common scenario of stations accessing the Internet, since the access point is also able to bridge communication over to a wired network. Bridging between 802.11 and 802.3 networks, specified in the standard IEEE 802.1D, is facilitated by the use of common addressing and similar frame formats.

By default, the MAC protocol of 802.11 is completely distributed, wherein even an access point has the same priority and medium access procedure to follow as any regular station. This is is called the Distributed Coordination Function (DCF). An alternative access procedure, Point Coordination

Function (PCF), allows an access point to be in control of the medium and give stations permission to transmit via a polling mechanism. The PCF is standardized, but in practice it is not widely implemented or deployed.

In 802.11 terminology, a set of stations sharing the same medium (i.e., same radio channel, all in range of each other), are said to form a Basic Service Set (BSS). When in infrastructure mode, it is also possible to interconnect multiple APs via what is called a Distribution System (DS), to allow stations associated with different APs to communicate with each other, thanks to the APs relaying the MAC frames, forming what is termed a Extended Service Set (ESS). The DS can be any technology that interconnects the APs while following certain requirements; a classic example is a switched Ethernet network to which the APs are connected.

2.1.3 UMTS (3G cellular network)

UMTS is a major 3G wireless technology offered by telecommunication operators in licensed frequency bands ranging from 800 MHz to 2.6 GHz. It achieves wide coverage through spatial (cellular) reuse of radio resources, and roaming between operators. UMTS allows fast handovers between cells, and vehicular communications up to 250 km/h. The UMTS physical layer is based on *Wideband Code Division Multiple Access* (WCDMA), which can multiplex data from/to multiple terminals by applying different spreading codes to each terminal. UMTS Rel99 enables data communications with peak data rates ranging from 384 kbit/s (urban outdoor). HSDPA (Rel5) can provide data rates up to 14.4 Mbit/s, with a latency of about 70 ms. New radio techniques are expected to enable higher data rates for these technologies: UMTS LTE (Long Term Evolution) Release 8 offers nearly 300 Mbit/s for downlink and 75 Mbit/s uplink [7], while the next evolution, LTE-Advanced targets over 1Gbit/s of bitrate. The UMTS technology is strongly operator-oriented, meaning that only large telecommunication operators can afford to deploy UMTS cell networks, since they require costly licenses to exploit the required frequency spectrum, and the infrastructure equipment is also relatively expensive.

In UMTS, the connection of a terminal to the network is represented by an abstract entity called *Primary PDP Context*. A terminal can have multiple secondary PDP Contexts for the same network, with different QoS requirements, and the RRM (Radio Resource Manager) in the network tries to enforce those QoS requirements. However, only one Primary PDP Context is allowed for each network the terminal is connected to. The Primary PDP Context contains the base connection information, such as address

type (IP or PPP) and address. It should be noted that IP data traffic in UMTS terminals is exchanged directly over the PDP context; there is no 802-style MAC identifier involved.

2.1.4 WiMax (IEEE 802.16)

The IEEE 802.16 technology addresses the WMAN (Wireless Metropolitan Area Network). This technology tries to address the transmission range limitations of 802.11. In line of sight transmission band (10-66 GHz), an 802.16 cell can have a radius of 50 km, with a maximum data rate of 75 Mbit/s. It is used to offer access to Subscriber Stations or as a back-haul network. The IEEE 802.16a amendment enables the use of non line of sight transmissions in a frequency range from 2 GHz to 11 GHz, in both licensed and license-exempt bands. In addition to the single-hop point-to-multipoint operation, 802.16a specifies the support for mesh networks. The 802.16e amendment [8] adds mobility support, providing support for handoffs to stations moving up to vehicular speeds. The real 802.16e performance is considerably less [9] than the theoretical limits, with total bandwidth per base station averaging a few Mbit/s or less, and performance decreasing considerably for distances above 1 km. New radio techniques, included in IEEE 802.16m, are expected to enable higher data rates: 100 Mbit/s and 1 Gbit/s for mobile and fixed stations, respectively.

There are three types of 802.16 equipments: *base station*, *subscriber station*, and *mobile station*. Subscriber and mobile stations represent fixed and mobile end user terminals, respectively, while base stations are attached to transmission towers and connect the WiMax network to an infrastructure. The MAC protocol of 802.16 is completely controlled by the base station and, unlike 802.11, the base station controls the time periods in which subscriber stations are allowed to transmit. In 802.16 MAC protocol, time is divided into periodic intervals, called *frames*. Each frame is further divided into uplink and downlink *subframes*, and inside there are *bursts*. The base station periodically transmits a *map* of the subsequent frames, indicating which stations are allowed to transmit on each burst. Thus, the access control is neither distributed nor polling based, making it very efficient. In addition, QoS concepts are built into the 802.16 MAC layer: the service is connection oriented, each subscriber station can have multiple connections to the base station, and connections have an associated set of QoS attributes.

In 802.16, MAC addresses compatible with 802.3 and 802.11 are used, and so bridging 802.16 to the other 802 technologies is possible and simple by following IEEE 802.1D and the so called *learning bridge* operation.

2.1.5 DSRC / IEEE 802.11p

Recently, a number of related standards are being developed to address vehicular communications. In the United States they are called Dedicated Short Range Communication (DSRC) [10, 11], and it includes spectrum allocation, MAC/PHY layer (802.11p), among others. In this architecture, two main types of equipments are defined. First, Roadside Units (RSUs) are devices that are placed regularly along roads at fixed locations, possibly connecting to an infrastructure. Second, On-board Units (OBUs) are devices inside the vehicles. Communication between two OBUs is called vehicle-to-vehicle (V2V) communication, while communication between an OBU and an RSU is termed vehicle-to-infrastructure (V2I) communication.

The DSRC frequency band allocation and spectrum channel allocation is specifically designed for vehicle-to-vehicle (V2V) or infrastructure-to-vehicle (I2V) communications. The 5.9 GHz frequency has been selected due to its propagation characteristics — up to 300 or 1000 meters range — which are suitable for vehicular communications. The main focus of DSRC is safety applications, and it allows several channels of communication, some of which are dedicated exclusively for safety applications, this way preventing commercial applications from accidentally interfering with higher priority safety applications. The 75 MHz spectrum band is divided into seven 10 MHz channels: one *control channel* (CCH), exclusively used for safety applications, and six *service channels* (SCH), that may be used for non-safety applications. Since the DSRC devices only have one radio, they are required to continuously hop between channels, in a synchronized fashion.

The 802.11p draft standard, Wireless Access for Vehicular Environments (WAVE), provides a MAC/PHY layer for use in vehicular environment. It is based on modifications to 802.11a. The first difference is that 802.11p has to use 10 MHz while 802.11a uses 20 MHz channels. Thus, the data rates available to 802.11p are half of the ones in 802.11a, 3–27 Mbit/s instead of 6–54 Mbit/s. Additionally, 802.11p has a longer guard period, which provides better resistance against multipath error. Since 802.11p uses a licensed (but free) band, it has less interference from electronic equipment than 802.11a/b/g operating on the ISM band. The MAC protocol of 802.11p is based on the regular 802.11 DCF, but has some modifications. When tuned to a CCH, the RTS/CTS mechanism is not used, to reduce the overhead. Another modification, driven by privacy and tracking concerns, is that OBU devices periodically generate a new random MAC address. When a collision is detected in a generated MAC address, a new address is randomly generated, and the process is repeated until one is found that is not

	802.11 a/b/g	**Fixed 802.16**	**Mobile 802.16**	**UMTS**	**802.11p**
Coverage	Local	Metropolitan	Metropolitan	Wide	Local+ (300–1000) m
Max. Data Rate (Mbit/s)	11, 54	75	75	0.384, 2 (R99) 3.6, 7.2, 14.4 (HSDPA)	27
Band (GHz)	2.4 (ISM), 5 (U-NII)	2–11, 10–66	2–11	<3	5.9
Licensing	Unlicensed	Both	Both	Licensed	Licensed
Mobility	Pedestrian	Fixed	Vehicular (<120 km/h)	Vehicular (<250 km/h)	Vehicular (<200 km/h)
Handover	Yes	No	Yes	Yes	Yes
Latency	Low	Low	Low	High (R99) Low (HSDPA)	Low
Mesh	Yes	Yes	Yes	No	No

Table 2.1: Main wireless technologies

used. There are also simplifications in the way that BSS is handled; 802.11p allows OBUs to communicate without associating to an AP (as in infrastructure mode 802.11) or even forming an adhoc BSS. To support this, the BSSID field of 802.11p MAC frames can be set to a "wildcard" value (all ones), and the ToDS and FromDS fields can be both set to zero. Thus, there is no time wasted on AP association.

2.1.6 Summary

In Table 2.1 the main wireless technologies used to build local, metropolitan, and wide area networks are identified. In WNMTs, there is no single link layer technology satisfying all their needs, and multiple technologies have to be combined. To connect vehicles to the infrastructure, 802.11a/b/g are generally not adequate due to its very short range and low maximum supported speed. For this purpose, technologies such as mobile 802.16 and UMTS are better suited. Between 802.16 and UMTS the choice is not trivial, but deciding factors may include network deployment costs, traffic costs, and the strategic requirement of some transport operators controlling their entire network. For 802.11p to be effective alone, the density of RSUs would need to be very high, resulting in a costly network. On the other hand, if the RSU density is low, 802.11p can still be used as alternative, in parallel to 802.16/UMTS: where 802.11p is available, vehicles can use it to obtain better bandwidth with less cost, otherwise they fall back to 802.16/UMTS. Finally, WNMTs may benefit from having a core based on

gigabit ethernet or fixed directional 802.16.

For providing Internet connectivity to the users' terminals, one could argue that a simpler and more robust solution is to allow users to connect via 3G network directly. But there are several advantages for end users to connect via WiFi instead of 3G. The most important one is energy consumption. The paper [12] concludes that *"WiFi consumes one-sixth of 3G's energy"*. The main reason for this difference is simply transmission range: clearly, a device that needs to transmit to an antenna that is far away requires a lot more power. We could apply the same rationale for deriving similar conclusions for the 802.16 case, although we could not find any study comparing 802.16 and 802.11 energy efficiency in the literature. Additionally, WiFi is increasingly becoming a "universal wireless" *de facto* standard. While mobile equipments currently support a variety of cellular wireless technologies, most equipments support at least WiFi. Moreover, even if a user has a 3G contract that he can use, while roaming in another country the 3G standard he uses may not be supported in that country, or the cost of data while roaming is prohibitive. To conclude, although several technologies can apply for interconnecting the mesh network nodes, for the end user terminals 802.11 is clearly the best choice, if not the only one.

2.2 Networking technologies

The term "networking" describes a process by which nodes that are not directly connected via a link-layer technology (not in range) can communicate with the help of additional intermediate nodes, which relay information, thereby extending the communication range. Typically, the networking function can be split into two separate parts: forwarding and routing.

Forwarding is the process by which a network node receives a packet in one interface and re-transmits the same packet in another or same network interface, while modifying some packet headers. The purpose of packet switching is to allow two distant nodes to communicate even though they are not directly connected, by using the packet forwarding services of additional intermediate nodes. The term that applies to those intermediate nodes varies according to the technology. For instance, in IEEE 802.1D networks, they are called *bridges*, while in IP based forwarding they are called *routers*. In this section we examine the most common, and relevant in the context of this thesis, packet forwarding technologies: IEEE 802.1D, IP, and MPLS.

Routing, on the other hand, is the process by which nodes compute/acquire routes, which are basically packet forwarding instructions, telling the

node information needed to re-transmit a packet so that it reaches its intended destination. Usually these routes are discovered via a *routing protocol*, by which nodes exchange information about network topology. Some routing protocols can run periodically and decoupled from packet forwarding — these are called *proactive routing protocols* — or they can run only when a packet needs to be forwarded for a destination and the route for that destination is not known — these are called *reactive routing protocols*.

2.2.1 802.1D bridging

The most simple of the packet forwarding algorithms is provided by the 802.1D standard, which describes how IEEE MAC Bridges should forward packets. The 802.1D standard describes what is informally known as the "learning bridge". The learning bridge has a set of *ports*, to which 802.3 stations, or other bridges, are connected. Because 802.3 is not connection-oriented[2], when the learning bridge is first started it does not know which stations (MAC addresses) are connected to which port. When an Ethernet frame is received in one of those ports, the bridge re-transmits a copy of the frame through the port to which the destination station (identified by the destination MAC address) station is attached. If the learning bridge does not know the correct output port, it simply floods the packet through all ports except the incoming port, thereby ensuring that at least one copy of the packet reaches the intended recipient. In any case, every time a learning bridge forwards a frame, it learns from the source address of each frame the location of the corresponding station. Thus, over time it builds a forwarding table, which associates MAC addresses with bridge ports. Entries in this table are soft-state: if they are not refreshed periodically, they expire (after 300 seconds, by default) and are removed.

Although the basic Learning Bridge algorithm can be considered at the same time forwarding and routing, since routes for MAC stations are being discovered at the same time as packets are being forwarded, it only works when 802.1D bridges are connected in a tree topology. Because bridges sometimes have to flood an incoming frame through all output ports, the existence of forwarding loops between bridges can cause a network collapse. For instance, consider a MAC frame with destination address of a station that is no longer connected to the bridge. In any bridge the packet passes through, it does not know the location of the destination MAC station, and so it floods the packet, eventually creating an endless forwarding loop. This

[2]The 802.2 Logical Link Control (LLC) layer above may be connection-oriented, but it is rarely implemented [13].

problem is aggravated by the fact that 802.3 frames do not have a TTL. To prevent this problem, the 802.1D bridges run a Spanning Tree Protocol (STP) which detects forwarding loops and disables the use of certain bridge ports to avoid them, forcing the network topology to become a tree.

2.2.2 IP networking

Basic IP forwarding

IP routers are nodes that have a set of interfaces, each interface has at least one *IP address*. The IP address, in its IP version 4 variant, is usually represented in the form of four numbers between 0 and 255 separated by dots, e.g. `1.2.3.4`. In IP networking, two interfaces of two nodes can exchange packets directly only if they have a physical link between them and appear to be in the same LAN because they share a common IP address prefix. The common prefix is termed *network address*, and is usually represented by address/prefix-length, e.g. `192.168.1.0/24`. The basic IP packet forwarding process takes the following steps:

1. The packet is received in a network interface;
2. The layer-2 header is removed and IP header parsed;
3. The packet time-to-live (TTL) counter, in the IP header, is decremented, packet is dropped if TTL reaches zero;
4. The destination address is looked up on the IP forwarding table, obtaining a pair of values: output interface, and next hop IP address;
5. The packet is re-transmitted via output interface toward the "next hop"; this transmission is a L2 transmission, and usually involves adding a new L2 header.

As seen the above steps, a fundamental input to the forwarding algorithm is the forwarding table, also known as Forwarding Information Base (FIB). In IP routers, the forwarding table contains as key a list of IP addresses, and as values output interface and next hop IP address. As an optimization, IP routers also support network addresses as keys in the forwarding table, allowing the forwarding table to become extremely condensed. The process that looks up individual IP addresses in the table considers a match any IP address that has the same network prefix as a network address in the table.

The forwarding tables can be configured manually, but usually are automatically configured via a *routing protocol*, such as Open Shortest Path First (OSPF) [14] or Border Gateway Protocol (BGP) [15].

Shortest Path First (OSPF) routing

Open Shortest Path First (OSPF) [14] is a *link state proactive* routing protocol for IP networks. Routers that run OSPF are able to automatically discover shortest paths, or routes, for all the IP addresses (usually networks) that any router owns. For instance, if a router A has an interface that is attached to a LAN segment with IP network address `10.0.1.0/24`, router B can discover a path to reach that network, with router A being the last hop of the discovered path. To enable such discovery, all routers periodically exchange messages with each other with the topology information, hence the "proactive" term, to denote that the routing protocol proactively discovers all the routes, even before they are actually needed to forward a packet. The information exchanged consists basically in each router reporting to all the other routers a list of its neighbors and metric (cost, distance) to reach those neighbors. In other words, the routers report the state of its links to its neighbors, hence the term "link state" being used. When a node receives the link state information from every node in the network, it builds a graph data structure and runs Dijkstra's Shortest Path [16] algorithm to obtain the shortest paths.

OSPF sends link state advertisements triggered by network changes. These advertisements are limited to be broadcast at most every 5 seconds, and at least refreshed once every 30 minutes. In any case, link state updates sent by one node reach all other nodes, frequently flooding the entire network with OSPF control packets. This approach does not scale well for large networks (60–80 nodes) with frequent topology changes. For better scalability, OSPF supports the concept of *areas*: the network administrator can split the network into smaller network partitions, called areas, and essentially different OSPF instances run in different areas. Although this approach can significantly improve scalability, it comes at the cost of some degree of engineering and configuration required to assign each router to a specific area. Moreover, this approach works poorly with highly dynamic networks, such as vehicular networks, where it is not efficient to assign a moving node to a single static area while it moves across the entire network diameter.

Optimized Link State Routing (OLSR)

Optimized Link State Routing (OLSR) [3] is routing protocol tailored to IP adhoc networks. Like OSPF, it is a proactive link state routing protocol, but instead of supporting network segmentation it employs an optimization for

link state dissemination wherein only a subset of the nodes, called *multipoint relays* (MPRs), are tasked with relaying the updates.

The following message types are defined in OLSR:

HELLO: this message is transmitted periodically (every 2 seconds by default) by OLSR nodes, but is not relayed by nodes that receive it. Its main purpose is to let OLSR discover its neighbors. Besides the IP address of the originating node, the HELLO message contains a list of IP address that this node has discovered as direct neighbors. In this way, a OLSR node, when receiving HELLO messages from its neighbors, discovers not only the presence of those neighbors but also the list of neighbors of each neighbor; these are called *two-hop neighbors*. The two-hop neighbors set is used for MPR selection purposes;

TC: the TC (Topology Control) message is generated periodically (every 5 seconds by default) and contains the topology information (i.e. link state) of each node. TC messages are to be forwarded by the MPR nodes so that it floods the entire network;

MID: the MID (Multiple Interface Declaration) message enables OLSR to work with nodes containing multiple interfaces. Since, in IP networks, each interface necessarily needs a unique IP address, and since HELLO messages necessarily contain as source address the IP address of the interface in which it is transmitted, OLSR nodes in the network need to find out if multiple HELLO messages come from the same node, for path computation purposes;

HNA: the HNA (Host and Network Association) message allows nodes to inject non-OLSR route information, to be disseminated in the OLSR network. This could be used, for instance, to advertise non-OLSR nodes or networks that are attached to an OLSR node.

The MPR concept is unique to OLSR[3] and is the main reason for the "Optimized" claim in the protocol name. It works as follows. First each node acquires a set of neighbors and two-hop neighbors through HELLO message exchanges, as previously explained. Then each node selects a subset of its neighbors as MPRs so that all the two-hop neighbors can be reached by at least one MPR. The MPR selection is completed when each node advertises this choice to each of the nodes it selected as MPR; this is also done

[3]It is innovative in adhoc routing protocols, although the concept was borrowed from the wireless technology HIPERLAN.

via a HELLO message. In the future, any message that a node transmits and which is to be flooded is only re-transmitted by those nodes that were selected as MPRs of that node. This way, the number of transmissions is reduced without compromising the effectiveness of the flooding process.

The MPR flooding process is able to reduce the overhead of TC flooding to 15–40% of the classical full link state flooding, depending on the node density [17]. This overhead reduction is due to only MPR nodes relaying link state information (TCs), while in a typical link-state routing protocol, such as OSPF, all nodes do this task. However, there are several good reasons for not copying OLSR's MPR flooding technique.

First we have to realize that the effectiveness of MPR flooding overhead reduction is highly dependent on the average node density, i.e. the average number of neighbors of each node: higher node density leads to a higher benefit of using MPRs. However, when designing a protocol for vehicular networks, which can be considered sparse[4], the benefit of MPRs would be small. There are also computational complexity issues. Finding an optimal MPR set is an NP-complete problem [18]. In OLSR, an heuristic to computing a good (but not optimal) MPR set is used, but it still has a computational complexity $O\left(3\Delta M + \Delta\right)$, considering that Δ represents the maximum number of one-hop neighbors, and M represents the maximum number of MPRs selected by a node [19]. We can apply the approximation that $M \approx 5\sqrt[3]{\Delta}$ in a random unit graph topology[20], and find that computing MPR set using the heuristic will involve 560 steps per received HELLO, for 4 neighbors, or 5000 steps for 10 neighbors. This means 2500 steps per second for 10 neighbors, using the default HELLO interval (2 seconds). Clearly, the computational complexity of MPR selection is not negligible, and it must be considered as a factor against this method. The use of MPRs can negatively impact the effectiveness of the routing protocol to adapt to node mobility. In [21] it is stated that *"We see that MPR needs on average 7 iterations before being able to provide OLSR with accurate topological data. [...] If we consider mobility, every time the topology is changed, OLSR loses between 3 to 4 seconds before being able to reorganize its routes"*. This contrasts with the 0–2 seconds needed to receive a HELLO from a new neighbor in the case when no MPRs are needed. The use of MPRs can also have an impact on the robustness of the flooding process in the presence of link failure. A simple scenario is shown in [22] where some topology information is lost with only a 45% link failure probability when MPRs are used, compared

[4]Due to mobile WiMax range and capacity limitations, the node density in the WiMetroNet scenario is expected to be relatively low.

to 85% for the blind flooding case.

In conclusion, the use of MPRs improves the scalability of OLSR, when compared to a pure link state routing protocol, but the scalability improvements are low and the cost, for instance in terms of computation complexity, is high. In any case, in OLSR the TC messages are generated frequently and flooded through the entire network, therefore it does not scale well to large networks.

Ad hoc On Demand Distance Vector (AODV)

The Ad hoc On Demand Distance Vector (AODV) routing protocol [4] is a reactive, or on demand, routing protocol. It does not periodically exchange messages for route discovery; instead, it only attempts to discover a route when the router is requested to forward a packet and it does not know the route to the destination address. To discover a route for a certain destination, AODV defines two message types, Route Request (RREQ) and Route Reply (RREP). When a node wants to discover the route to a destination, it broadcasts a RREQ packet containing the destination address[5]. The RREQ is forwarded by every node that receives it, consecutively, until it reaches the destination node, which stops the flooding. Each node that received a RREQ creates a *reverse path* entry which records the path to the source node. The destination node replies to a RREQ with a RREP packet, which travels via the reverse path entries that were create by the previous RREQ message. The RREP eventually reaches the source node, thus providing the source node with a confirmed route to the destination.

When an AODV node is part of an active path, and new paths are not being negotiated, normally no more control messages are required as long as data packets are being forwarded to keep the path active (otherwise, the path becomes inactive after 3 seconds). However, if the AODV node moves, a neighbor could still be sending data packets to it without realizing it is no longer in range. In other words, the link has been broken, but the upstream neighbor does not realize it. Using L2 notifications, it would be possible to detect the link failure. However, AODV, being a L3 protocol, has its own L3 based mechanism to detect the link loss, based on "hello" packets. If no other AODV control traffic exists between the nodes, AODV periodically broadcasts special RREP packets with TTL=1, called hello packets. Then, when a node's neighbor stops receiving control packets for a period of time, it assumes the link is broken. The sources affected by the link loss are

[5]In the RREQ message, the destination address of the packet is the broadcast address.

informed that the path is broken via Route Error (RERR) messages, to give them an opportunity to try to discover alternative paths.

AODV is a routing protocol that has the merit of not taxing the network excessively when the amount of traffic in the network is low. Because only user traffic triggers the flooding of control packets, if the network is very large but only a few nodes are communicating over it, then the routing overhead is very low. Moreover, the overhead remains low regardless of amount of traffic, as long as the number of communicating pairs remains the same. However, this overhead can become quite high when the number of nodes communicating increases. From an engineering and management perspective, a reactive routing protocol is difficult to predict and manage because it is completely dependent on user traffic patterns, therefore out of our control.

2.2.3 Multi-protocol Label Switching (MPLS)

The Multiprotocol Label Switching Architecture [23], is based on the core concept that great performance gains can be achieved if, when switching a large number of similar packets (eg. DiffServ aggregate flows), a path is first established and recorded across a network, and then all subsequent packets just follow the recorded path. In MPLS terminology, the set of packets that follow the same path is called *Forwarding Equivalence Class* (FEC), and the path itself named *Label Switched Path* (LSP). The name for the LSPs comes from the fact that, once a path has been established, packets entering an MPLS domain are classified into FECs and then they are assigned a label based on the corresponding FEC. Inside the MPLS domain, packets are transmitted with a small MPLS header, named *shim header*, that contains the assigned label and little more information. The MPLS labels always have local meaning for each node/port pair, thus packets' labels have to be swapped as they travel through the nodes of an LSP. The label switching algorithm can be summarized like this:

1. A packet arrives on an input port;
2. **if** packet contains an MPLS shim header **then:**
   (a) Look at the $(label_{input}, port_{input})$ pair, use a lookup table to map into a FEC;
   (b) Use another table lookup to determine the $(label_{output}, port_{output})$ pair from the FEC;
   (c) Swap $label_{input}$ for $label_{output}$ in the packet shim header;

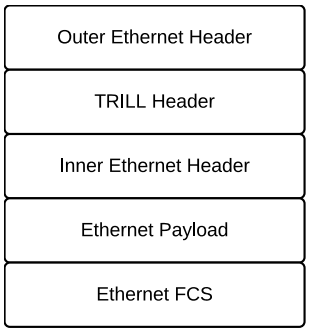


Figure 2.1: TRILL Encapsulation

(d) Queue de packet for transmission in $port_{output}$.

3. **else:**

(a) Use traditional IP-based routing.

Considering that the MPLS label is just a simple 20-bit integer value, it's easy to see that label switching is more efficient than L3 switching. But more importantly, MPLS allows *traffic engineering* to be deployed on high performance core networks with little or no performance penalty. This is due to the fact that complex matching rules for determining the FEC for packets may be placed on lightly loaded edge routers (the so called *Label Edge Routers*), while at the MPLS core the paths are already established and switching is based on MPLS labels as usual, meaning that the cost of switching best-effort and traffic engineered packets/flows is practically the same.

2.2.4 Transparent Interconnection of Lots of Links (TRILL)

The IETF Transparent Interconnection of Lots of Links (TRILL) [24] is working towards a standard solution for shortest-path frame routing in a multiple-hop 802.1-compliant network with arbitrary topology. For that purpose, TRILL proposes the concept of Routing Bridge (Rbridge), a node running the IS-IS [25] link-state routing protocol at L2. Other goals of the solution are: minimal configuration, routing loop mitigation (through the use of a TTL field) and legacy node support.

In TRILL, end terminal MAC frames are encapsulated by an Rbridge node, a TRILL header being added in the process. The encapsulated MAC frame, shown in Fig. 2.1, is then transported over a network of Rbridges,

using the TRILL header for routing, until it reaches the Rbridge to which the destination MAC address is connected. At that point, the encapsulation headers are removed and the original MAC frame is transmitted to the target LAN, finally reaching the destination MAC station.

The main fields of the 48-bits wide TRILL header are: 1) a 6-bit *Hop Count*, 2) a 16-bit *Egress RBridge Nickname*, and 3) a 16-bit *Ingress RBridge Nickname*. The RBridge "nicknames" are numeric identifiers the uniquely identify each RBridge within the network. These nicknames can be manually configured, or randomly generated, but the routing protocol allows duplicated nicknames to be detected and avoided. In TRILL, the Egress RBridge Nickname indicates the destination RBridge that an encapsulated packet must reach. The Ingress RBridge Nickname indicates the RBridge from which the packet entered the TRILL network.

In TRILL, the IS-IS routing protocol is used to discover routes between Rbridges. The IS-IS protocol is similar in concept to OSPF, but is not tied to IP networks, and so was adapted in TRILL to work at Layer 2. Nonetheless, it has the many of the same limitations of OSPF, in particular scalability and support for dynamic networks / mobility. Moreover, TRILL targets maximum compatibility with 802 bridged networks and does not limit the forwarding of broadcast frames in any way, which leaves it with limited scalability.

2.2.5 802.11s mesh networking

The 802.11s [26] is a Draft standard under development that aims to dote the 802.11 MAC layer with enhancements to support wireless LAN mesh topologies. With 802.11s, 802.11 stations will be able to form a mesh, and perform over-the-air multi-hop packet forwarding among themselves, without the aid of a wired Distribution System (DS). The set of nodes that directly participate in the 802.11 mesh network is said to form a mesh BSS. In 802.11s, we can have the following types of nodes:

mesh station: Any node that has a 802.11 interface in mesh mode;

access point: A node that has a 802.11 interface in infrastructure mode, and is also connected to the DS via an additional non-802.11 interface (typically 802.3);

mesh gate: a node that bridges a mesh BSS with a DS;

portal: a node that bridges a DS with a non-DS network. Typically, a *portal* is effectively an Internet Gateway.

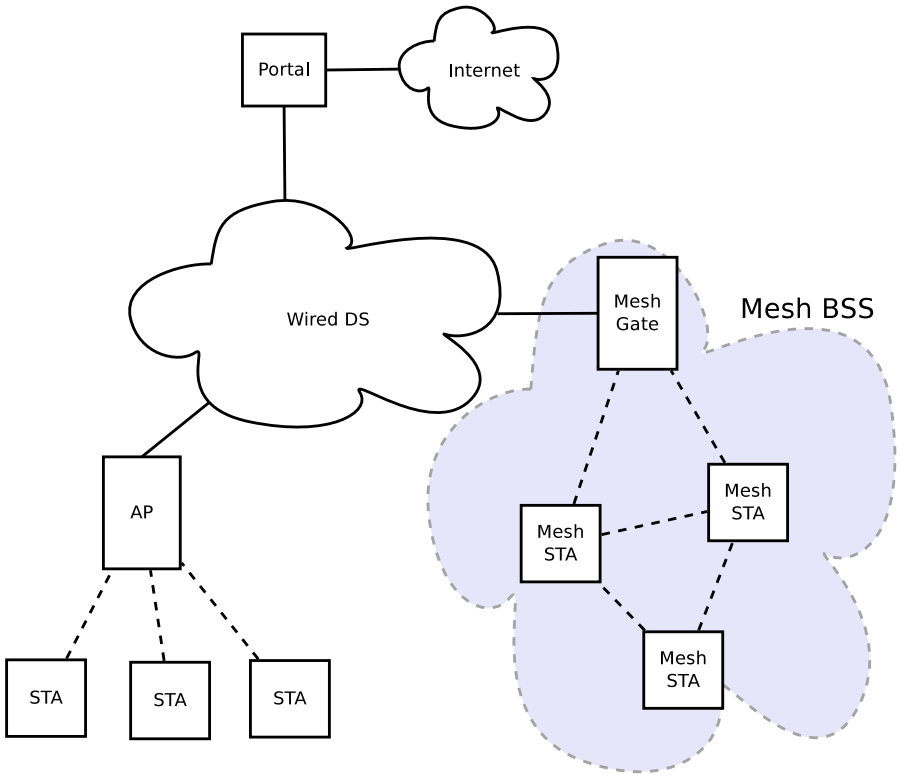


Figure 2.2: Example mesh and infrastructure BSSs.

These roles are exemplified in Fig. 2.2. The same node may be able to have multiple of the above roles co-located. Moreover, 802.11s allows a mesh BSS to become a DS for an ESS. Thus, 802.11s offers great flexibility for topologies and deployment options.

The 802.11s amendment defines an additional 802.11 header field named *Mesh Control* (MC). Inside the MC field, up to two additional MAC addresses can be carried. A total of six addresses are used in order to support the use case of a mesh network operating as DS; the mesh network has to relay infrastructure mode MAC frames, and therefore needs the additional two addresses for encapsulation purposes. Additional MC sub-fields include a time-to-live (TTL), and Mesh Sequence Number, among others. These are used to discard packets in forwarding loops and duplicated packets, respectively.

For path selection (i.e., routing), 802.11s defines the Hybrid Wireless Mesh Protocol (HWMP). HWMP is said to be a hybrid routing protocol because it combines both reactive and proactive modes of operation. The reactive, or on-demand, mode of operation is always available and is inspired by the AODV routing protocol; for path discovery, it uses the messages Path Request (PREQ), Path Reply (PREP), and Path Error (PERR). The proactive mode can optionally be used in addition to the reactive mode, and it consists in a tree that is created by a *root node*, which periodically broadcasts Root Announcement (RANN) messages; other nodes listen to RANN messages, record a path to that root node, and re-transmit the message for other downstream nodes to hear, as well as send a PREQ message upstream to the root node, so that the root node also knows the reverse path to each intermediate node. The proactive tree that is built allows all nodes to always know the path to a root node, and vice-versa. This allows any two nodes to communicate by relaying traffic through the root node. Two nodes can also use the PREQ/PREP messages to find out more direct paths of communication that do not necessarily involve the tree root node, for greater efficiency.

The scalability of 802.11s is quite limited for essentially two motives. First, it inherits the weakness of AODV, namely that the routing overhead increases with the amount of user data. Second, is has to honor the 802 service model and transport all traffic, even broadcasts; “broadcast storms” are allowed and become a problem for even just medium sized networks.

2.2.6 Summary

In order to forward a packet based on its destination address, each switch has to look up a forwarding table. Each entry of this table associates a destination address to an output port. Technologies such as 802.1D behave in this way and, since they use unstructured 48 bit MAC addresses, their forwarding tables contain one entry for each known address. Some other network technologies, such as traditional IP, may use structured addresses (the network address); in this case, an entry of the table contains forwarding information for an entire sub-network. Structured addresses lead to shorter forwarding tables and shorter lookup times than unstructured addresses; they also enable the deployment of large networks, because entries in the forwarding tables can be organized using techniques such as classless IP addresses. However, structured addresses preclude network auto-configuration because they have to be engineered and configured manually. To solve the auto-configuration problem, IP adhoc protocols, such as OLSR and AODV, have been proposed. However, they are not transparent to the end user terminal, and do not scale for medium/large networks (more than one hundred nodes is impractical with unmodified OLSR or AODV).

A virtual circuit is commonly interpreted as a path along the network which must be established before packet transmission. This path may be characterized by a single identifier, with global significance, or by a sequence of identifiers (labels), each having local significance between adjacent switches. When a switch receives a packet, it looks up its forwarding table. An entry of this table contains information about the output port and about the label that will be used by the packet on the next segment of the path. Thus, from switch to switch the packet sees its label substituted. Technologies such as the Multi-protocol Label Switching (MPLS) use labels as identifiers.

Some other networking technologies, such as tunnels, Transparent Interconnection of Lots of Links (TRILL), IP source routing, MPLS based VPNs, or 802.11s, use layered approaches, where packet switching and virtual circuit switching may be combined. In these cases, paths are defined based on MAC addresses, IP addresses, or MPLS labels. The 802.11s packet, for instance, contains four MAC addresses: source station, destination station, and the addresses of the mesh points attached to the current segment of the path. When an 802.11s mesh point receives a packet it looks up its forwarding table using the packet destination address; an entry of this table contains the MAC address of the switch to which the packet must be forwarded, which is used to re-arrange the MAC addresses of the packet before

it is forwarded. Additional addresses may be used when dealing with legacy stations or networks.

Networks should avoid loops, that is, a packet should not pass twice through the same switch. Technologies such as 801.1D prevent loops by computing a unique tree which interconnects every switch. Other technologies, such as IP, MPLS, 802.11s, or TRILL, enable each switch to compute its, possibly shortest, path tree to the other switches, thus enabling a better usage of the network resources. When the topology of a network changes the trees have to be recalculated, and this is a distributed process. In large networks the time required to recalculate the trees after a network modification may take seconds. During this time, temporary network loops may appear. In order to avoid this and other configuration problems, technologies such as IP, MPLS, 802.11s, and TRILL include in their packets a Time To Live (TTL) field; this field is decremented every time a packet passes through a switch and the packet is discarded when its value reaches zero, thereby ensuring that packets caught in loops are eventually eliminated. Thus, the existence of a Time To Live (TTL) field is intimately connected to the support of mesh topologies. Technologies such as MPLS, 802.11s, and IP have a TTL and support mesh topologies, while 802.1D has no TTL and does not support mesh topologies.

Candidate solutions for WNMTs should contain the following ingredients: scale to thousands of nodes, support mesh topologies, support mobility, and be auto-configurable. None of the technologies in Table 2.2 fulfills all of these requirements. While 802.1D is auto-configurable, it does not scale to a WNMT size, mainly due to the way it handles broadcasts [2], and does not support mesh topologies. TRILL adds a TTL and link state routing to 802.1D networks, allowing it to support mesh networking, but it improves little on the scalability of 802.1D, and does not support mobility. 802.11s supports mobility and mesh networking, but has very limited scalability. IP and MPLS are both scalable to a WNMT size and support mesh topologies, but their control planes require manual configuration. However, their data planes are useful, particularly MPLS due to its slightly lower complexity and higher flexibility.

2.3 IP based mobility

IP networks are ubiquitous, but IP(v4) was not designed with mobility in mind. The problem is that the IP addresses used in these networks have a double function. On one hand, they are identifiers of hosts and interfaces.

	802.1D	802.11s	TRILL	Trad. IP	Adhoc IP	MPLS
Structured addr.	No	No	No	Yes	No	Yes
Auto-config. addr.	Yes	Yes	Yes	No	Yes[a]	No
TTL	No	Yes	Yes	Yes	Yes	Yes
Scope of addresses	LAN	LAN	LAN	Global	LAN	Link
Network size	Hundreds	Tens	Hundreds	Millions	Tens	Thousands

[a] Several experimental address auto-configuration protocols proposed [27].

Table 2.2: Networking technologies, assuming common control planes

On the other hand, they are locators. For instance, to communicate with a host with address `192.168.10.3`, one could say that `192.168.10.3` is the identity of that host, allowing us to uniquely identify it. But at the same time, `192.168.10.3` "belongs" to the IP network `192.168.10.0/24`. While it is true that routers know implicitly the path to `192.168.10.3` because it is the same as the path to network `192.168.10.0/24`, this means that our end host cannot move from one router to another router without changing its IP address. But changing the IP address means that correspondent nodes (the nodes that are currently communicating with our node of interest) will no longer know how to reach the mobile node, and so communication will be broken, at least until resumed by higher layer protocols. A similar problem occurs with routers themselves. For each interface of every router, the IP address of the interface belongs to an IP network prefix, and the network prefix must be the same as the prefix used by the neighboring router, on the other side of the link. Thus, if a router moves, changing the physical topology of the network, it must adopt a different IP address in the network interface(s). Such a change will break communications, at least until the routing protocol discovers the new address and finds new paths.

Although IP was not designed with mobility in mind, to address the above problems several IP modifications and protocols, some of which at upper layers, have been proposed over the years. The main ones, according to active IETF Working Groups, are briefly presented in this section.

2.3.1 Mobile IP

The *Mobile IPv4* [28] solution has proposed to overcome the problem of IP networks using IP addresses as both locator and identifier. Using Mobile IPv4, mobile hosts may have two separate IP addresses — *Home Address* (HoA) and *Care-of Address* (CoA) — one (HoA) serving mainly as node identifier, while the other (CoA) serves as node locator. Mobile IPv4 defines

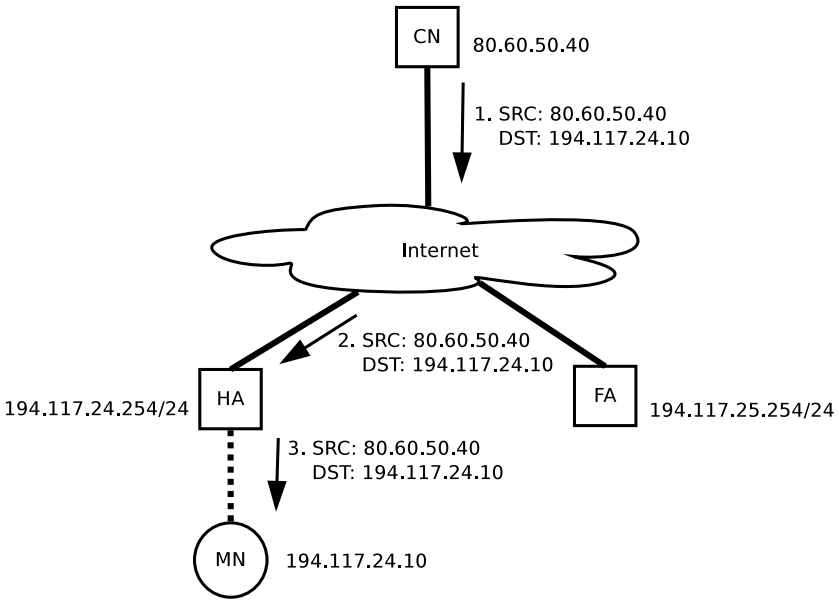


Figure 2.3: Mobile IP: mobile node at the home network

two types of infrastructure — *home agent* and *foreign agent* nodes that are designed to help with mobility aspects. The home agent is responsible for maintaining a registry of HoA↔CoA associations and, whenever the MN is known be visiting another network, tunneling packets destined for the MN's HoA to the new visited network. The foreign agent is a host located in each visited network; it takes care of de-tunneling packets sent by the home agent and transmitting those packets to the MN.

To understand how Mobile IP works, we may consider the scenario in Fig. 2.3, with a mobile node (MN) connected to its home network. Specifically, the access gateway of this mobile node also has home agent functionality. Another node is sending packets to the MN, let's call it correspondent node (CN). The CN sends packets to the MN's only address, `194.117.24.10`, and the packet is routed normally, without considering mobility. When the MN hands over to another network, as seen in Fig. 2.4, it becomes connected to a new router that is also a foreign agent (FA). Using the MIP protocol, the MN sends a *Registration Request* to the foreign agent, which forwards it to the MN's home agent. This registration contains, among other information, the HoA and CoA of the MN. A *Registration Reply* is transmitted by the HA, forwarded to the MN by the FA. Afterwards, when the CN sends a new packet to the MN, the packet is forwarded as

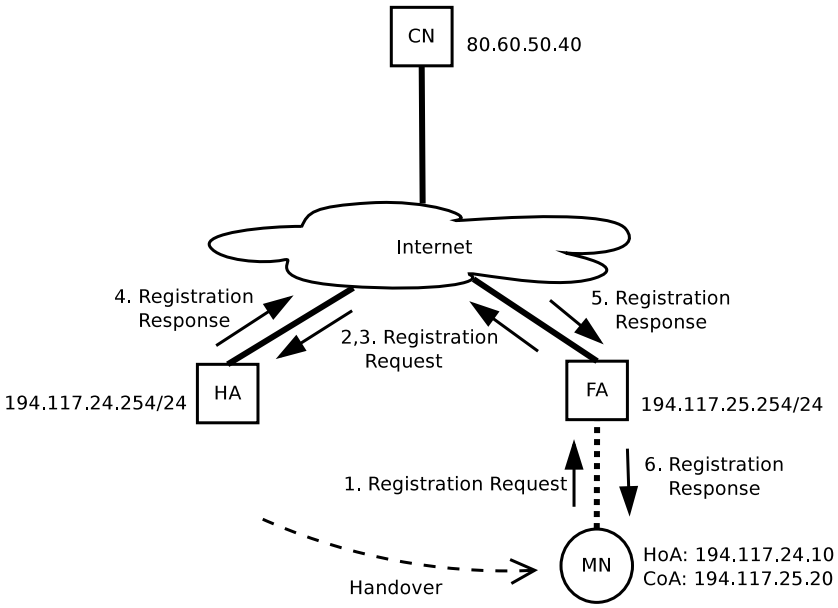


Figure 2.4: Mobile IP: handover registration

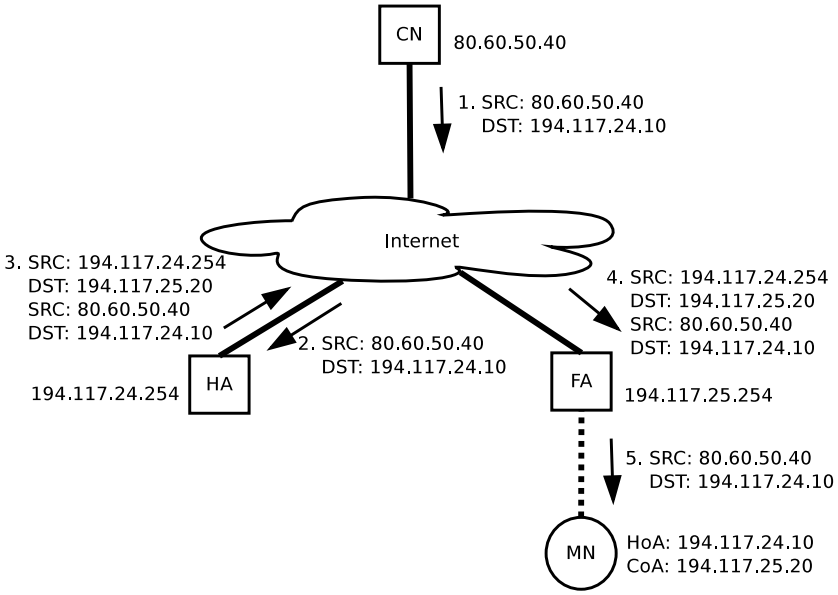


Figure 2.5: Mobile IP: correspondent node sends a packet after MN handover

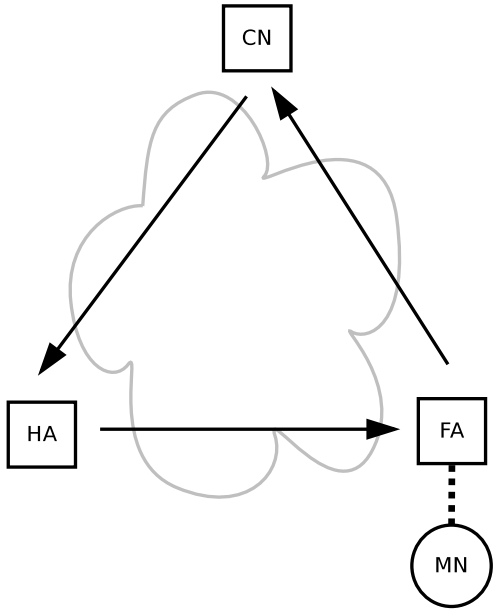


Figure 2.6: Mobile IP: triangular routing problem

shown in Fig. 2.5. The packet will be directed to the MN's home network, naturally passing by the HA. Since the HA has already been informed of the new CoA being used by the MN, instead of transmitting the packet directly, it tunnels it to the CoA. Since the FA is the gateway to the network to which the CoA belongs, the tunnelled packet passes by the FA, which de-tunnels it, i.e. removes the outer IP addresses, and the original IP packet is transmitted to the MN. The MIP protocol allows some variations of the above procedure, for instance the tunnel may end in the MN itself instead of the FA.

With Mobile IPv4, the packets from CN to MN go to HA and then to FA, while packets that the MN sends to the CN, assuming it is not mobile, are transmitted directly to it, following the most direct path[6], as shown in Fig. 2.6. This is called the triangular routing problem, and is a realization of the sub-optimality of the path that packets take from CN to MN when the MN is in a visited network, causing unnecessary traffic in the network and increased delay.

IP version 6 includes mobility support as part of the base protocol [29],

[6]This is not necessarily always true, as Mobile IPv4 supports the option of MN tunneling traffic to its HA, for location privacy reasons.

and it bears some similarities to the IPv4 version. The main difference is that *Binding Update / Acknowledgement* messages are used instead of Registration Accept / Response, and the FA is no longer needed. Additionally, Mobile IPv6 supports the so called *route optimization*, as follows. The MN sends packets to the CN via the direct path, but includes in the header not only it's CoA as source address but also the HoA in an extension header. The CN, when receiving the packet, will note the HoA↔CoA bindings, and from there on will start delivering packets directly to the CoA, bypassing the HA. Alternatively, the MN may also send an explicit Binding Update message to the CN, but only after completing the so called *return routability procedure*. The return routability procedure is a security measure that gives confidence to the CN that any future Binding Update messages received from the MN are authentic.

2.3.2 Mobile IPv4 Regional Registration

One of the problems in Mobile IPv4 is the need for the MN to register the new CoA with the HoA even if the HoA is very distant from the current node location, for instance in another country. In this scenario, the mobility signaling delay is increased due to the network distance to the HoA, with impact on the ongoing communication, such as severely delayed or even dropped packets. *Mobile IPv4 Regional Registration* (MIP-RR) is an experimental solution to solve this problem [30] by attempting to keep signaling caused by local mobility contained in that region. A new Gateway Foreign Agent (GFA) router function is introduced to manage the local mobility and shield the HA and CN from the effects of handovers within the site managed by that GFA. To accomplish this, the GFA intercepts and rewrites the CoA of the Registration Request messages that the MN initially sends to its HA, so that the tunnel that the HA believes to end in the FA instead ends in the GFA. Then, another tunnel is setup between the GFA and the real FA to which the MN is currently attached. When the MN hands over to a different FA within the same region (managed by the same GFA), it sends a *Regional Registration Request* message to the GFA, and the GFA↔FA tunnel is updated accordingly, although the HA and CN remain unaware of the modification. This is illustrated in Fig. 2.7.

2.3.3 Proxy Mobile IP

Proxy Mobile IP [31, 32] is a protocol and architecture for providing network based mobility management. In this architecture, the MN change between

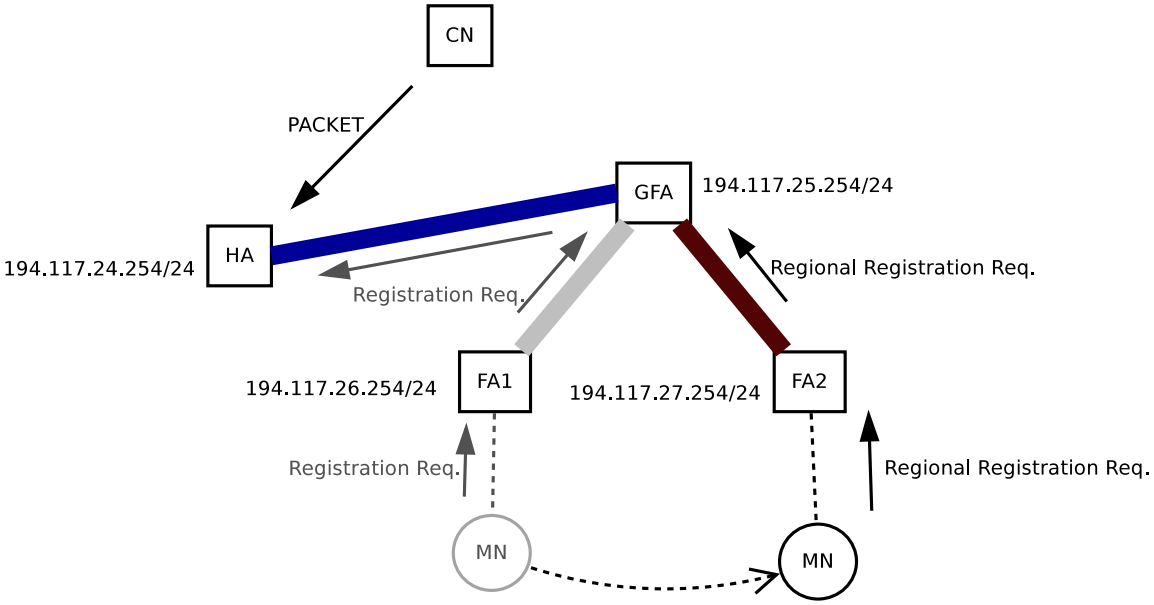


Figure 2.7: Mobile IP: regional registrations

different access gateways with full IP mobility (i.e. correspondent nodes are unaware of mobility), but without having to implement any kind of mobility protocol; the mobility is managed fully and transparently by the network, which is the main innovation of this proposal.

As shown in Fig. 2.8, in the Proxy MIP architecture at least one Local Mobility Anchor (LMA) exists in the network, and several Mobile Access Gateways (MAG). When a MN connects to a MAG, it acquires an IP (v4 or v6 or both) address by normal means, such as DHCP or Router Advertisement. The MAG then sends a Proxy Binding Update message towards the LMA, on behalf of the MN, and a bi-directional tunnel is setup between the LMA and the MAG. From there on, any packet addressed to the MN is transported over the tunnel, reaches the MAG, which then removes the encapsulation and transmits the packet over the link layer to the MN. When the MN hands over to another MAG in the same network, the new MAG acquires the MN's identifier (e.g. via MAC address), and issues a Proxy Binding Update. The respective Proxy Binding Acknowledgment will indicate the MN's previous address (HoA), so that when the MN tries to configure its mobile network interface address, the MAG will provide the same address as before.

Because the MN retains the same address during any handover, any

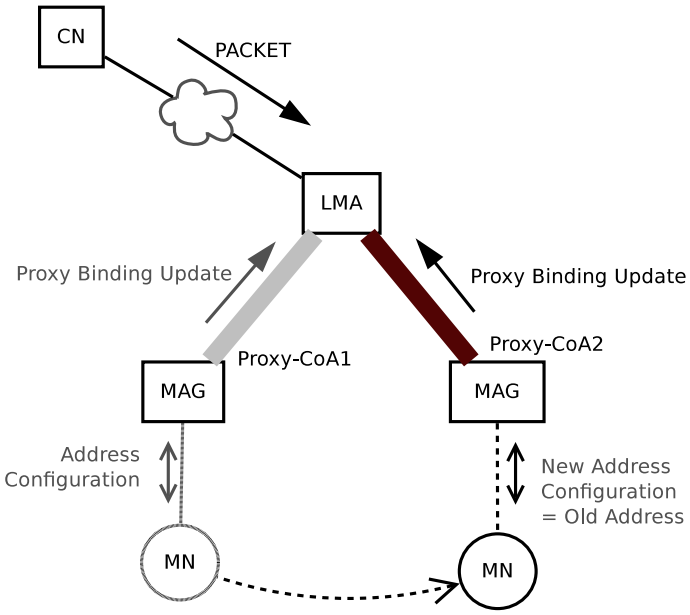


Figure 2.8: Proxy Mobile IP

nodes corresponding with our MN are not aware of the mobility. To the extent possible, even the MN itself is not aware of mobility, it only notices the connection to the infrastructure temporarily dropping.

2.3.4 Network Mobility (NEMO)

The IP based mobility solutions presented above attempt to solve the problems caused by the mobility of end user terminals. However, they do not handle the case of the infrastructure routers themselves moving, for instance in the case of a laptop giving access to a Personal Area Network (PAN), or a vehicle giving access to passengers' terminals. The Network Mobility (NEMO) Extensions for Mobile IPv4 [33] are a simple solution proposed to address this problem based on the concept of applying the MIP protocol not only to mobile terminals but also to mobile routers themselves. The NEMO protocol extends the normal MIPv4 Registration Request message with a new Mobile Network Request extension, allowing the mobile router to register a care-of address with its home agent. In this way, a bidirectional tunnel is set up between the home agent and the mobile router. Packets that are destined to the mobile router's HoA are instead tunnelled to the CoA location and this way delivered to the mobile router.

The NEMO approach is simple initially, but not very efficient due to the tunneling and indirect path. When there are multiple levels of routers which are all moving, NEMO causes multiple nested tunnels, increasing the hop count and delay of traffic, and reducing the path MTU. These problems are explored in [34].

2.3.5 Mobile IP Fast Handovers

One of the main problems of Mobile IP is the long connectivity interruption time it entails each time the MN changes point of attachment. When the MN associates with a new access point/router, it acquires a new address (CoA), and immediately the uplink packets can flow and reach the correspondent nodes. However, downlink traffic cannot be received by the MN until the full MIP signaling is completed, which includes Binding Update/Acknowledgment and return routability test procedure. These can take a few seconds to complete, in some cases, with adverse impact on time-sensitive communications, such as VoIP traffic. *Mobile IP Fast Handovers* [35, 36] attempts to reduce the handover time by allowing the MN to prepare the handover in advance for a New Access Router (NAR) while still connected to the Previous Access Router (PAR).

The protocol supports both MN initiated and network initiated handover; for the case of MN initiated handover, it works as shown in Fig. 2.9. The MN may notice the radio signal to the current AR (PAR) fading, and scanning could provide indication of a more powerful AP signal, candidate for handover. Using the Fast Handover protocol, the MN sends a Router Solicitation for Proxy Advertisement (RtSolPr) message to the PAR, soliciting L3 information about the AR that is associated to the discovered L2 AP. The requested information is delivered by a Proxy Router Advertisement (PrRtAdv). The MN may then decide to hand over to the NAR, and signals this intention by sending a Fast Binding Update (FBU) message to PAR containing the NAR identifier. Then the PAR communicates directly with the NAR, by sending a Handover Initiate (HI) to the NAR and receiving back a Handover Acknowledgment (HAck). The PAR then sends Fast Binding Acknowledgment (FBAck) message to both MN and NAR. As a result, the MN disconnects from PAR, and the PAR starts forwarding packets destined to the MN to the NAR. When the MN attaches to NAR, it sends a Fast Neighbor Advertisement (FNA) message to it, and NAR starts delivering packets to the MN. Now that the PAR→NAR packet forwarding is in place, it lasts for a few seconds in order to give the MN enough time to notify HA and CNs without risk of losing packets, after which time it is

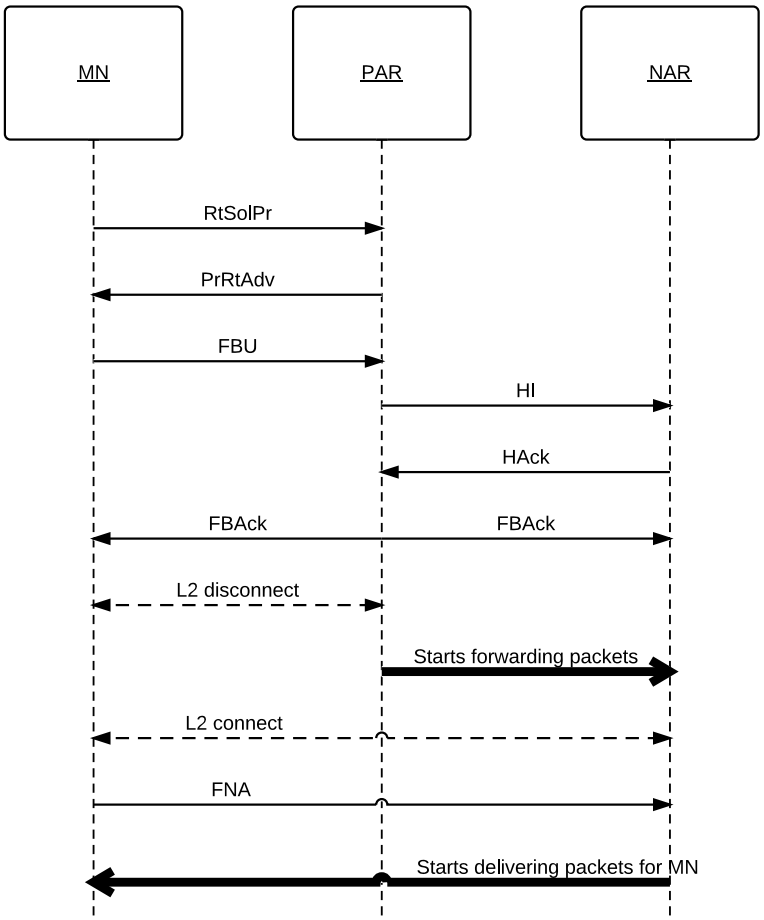


Figure 2.9: Mobile IP Fast Handover

automatically shut down.

The Fast Handover protocol approach is followed also by another competing, yet very similar, protocol *Low-Latency Handoffs in Mobile IPv4* [37]. Both protocols optimize handoff by allowing the MN to prepare the handover before disconnecting from the old AR, and redirecting traffic from the previous AR to the new AR for a period of time, while the MN handover registration signalling is completed, to minimize the number of packets lost during the transition. However, the time it takes to complete a registration remains unchanged by these approaches. With Fast Handover, a vehicle moving at high speed over small wireless cells will need to handover frequently, but Mobile IP signalling may still be taking place from a previous

handover when a new handover is required. A trail of tunnels would be created, and packets would follow an increasingly suboptimal path.

2.3.6 Host Identity Protocol (HIP)

The Host Identity Protocol (HIP) [38, 39] is an experimental architecture proposed to address mobility, security, and multihoming issues, by introducing a new layer between IP and transport layers, or L3.5. The main concept of HIP is that of introducing a new *host identity*, which is a global identifier for a host, and binding application sessions to this host identity, instead of directly binding them to IP addresses. In this way, HIP cleanly separates the roles of end-point identification and end-point locator. It is vaguely analogous to the the HoA / CoA roles in Mobile IP. Each host identifier (HI) is actually a public key, and they can be registered in a DNS server or Public Key Infrastructure (PKI). Since applications are already using HIs, enabling IPSec based security associations (for encryption or authentication) becomes straightforward. Moreover, an HI refers to a host stack as a whole, it can address any of a node's interfaces; handover between different interfaces of a host is facilitated, as well as handover between IPv4 and IPv6. Testing of experimental HIP and Mobile IPv6 implementations [40] appear to indicate that HIP can provide lower and more consistent handover latencies. To simplify and reduce the overhead of the protocol, a Host Identity Tag (HIT) is defined as a 128-bit hash of the full host identity. This HIT value is used by the protocol instead of HI wherever possible.

In order to communicate using HIP, the "initiator" host may use a conventional mechanisms, such as DNS, to obtain both the HIT and one or more registered IP addresses of a "responder" host. Then, initiator and responder have to perform an exchange consisting of four messages: I1, R1, I2, and R2, as shown in Fig. 2.10. This exchange forms a security association Diffie-Hellman style, after which the hosts acquire each other's IP addresses and can begin communication. In HIP, the PDU "UPDATE" allows a peer to change some parameter of the HIP association. This can be useful for mobility purposes, for instance. Fig. 2.11 shows as example a MN moving from one AR (pAR) to another (nAR), while receiving packets from a CN. When the MN connects to the nAR, it acquires a new IP address via normal mechanism (e.g. DHCP, Router Advertisement), and sends a HIP UPDATE message to the CN. The CN then updates the IP address for that HIP association, and the following data packets are addressed to the new MN location.

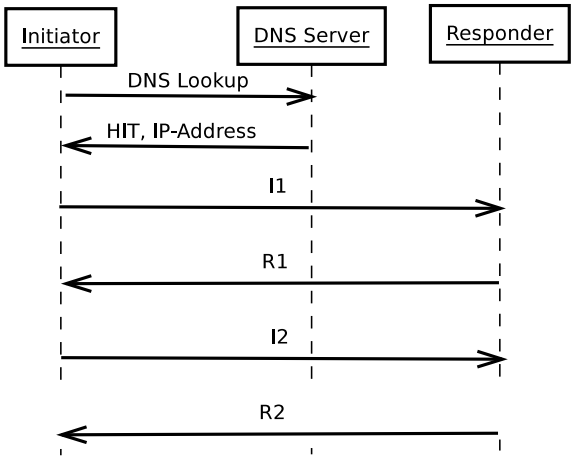


Figure 2.10: Host Identity Protocol: establishing an association

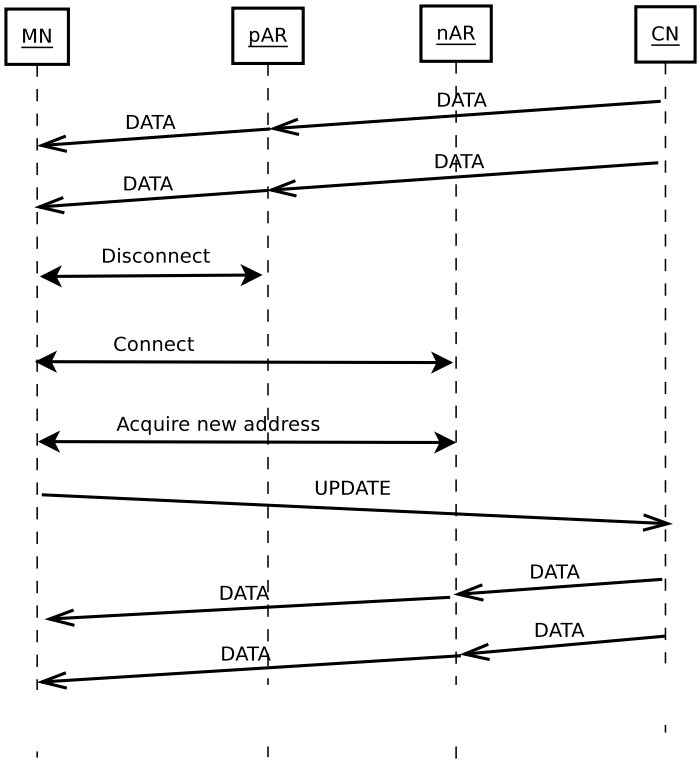


Figure 2.11: Host Identity Protocol: handover

2.3.7 Session Initiation Protocol (SIP)

We have shown how the mobility problem can be solved between layers 2 and 3.5, but these approaches are not without its faults. First, and foremost, mobility at lower layers requires modifications at network nodes, and deployment of such modifications takes too much time. This problem is compounded by the many choices for mobility that can be implemented, leaving network administrators unwilling to deploy any one until a clear "winner" emerges. Mobility, as defined as the ability for an application session to adapt to node mobility in timely manner, can also be implemented in the application layer itself. Perhaps the most notable application layer mobility approach is provided by the Session Initiation Protocol (SIP) [41].

In SIP, each user that can be contacted is identified by an email style URI, for instance `sip:user@domain.com`. In this URI, we can identify a user name and a domain name. The domain maps to an actual host name that implements a *SIP proxy*. A SIP proxy is a network server that keeps track of the location of each user in that domain. Users (or the users' terminals / applications) register themselves with the SIP proxy via a REGISTER SIP message. To start a session between two peers, one of the peers invites the other to a session by sending an INVITE message to the SIP proxy that manages the target peer. In the message, the initiator peer's IP address is included. The SIP proxy forwards the INVITE message to the target at its last known location. A "200 OK" positive acknowledgment flows in the reverse direction, in case the peer accepts the invitation, including its own IP address. When the MN hands over to a new PoA (AR), its IP address changes. To notify the communication peer of this change, a new INVITE is sent directly to the peer, allowing the session to resume (or restart, depending on the protocol) to adapt to the new IP address. Finally, the MN sends a REGISTER to its own SIP proxy, so that any future CNs may be able to contact it at the new address. This is shown in Fig. 2.12, which assumes both MN and CN are using the same SIP proxy, as simplifications. In the most general case, both users could be using different proxies.

2.3.8 Summary

Although IP was not designed with mobility in mind, to address the above problems several IP modifications and protocols have been proposed over the years. We can classify these proposals into two major classes of mobility management. Terminal mobility enables terminals to change point of

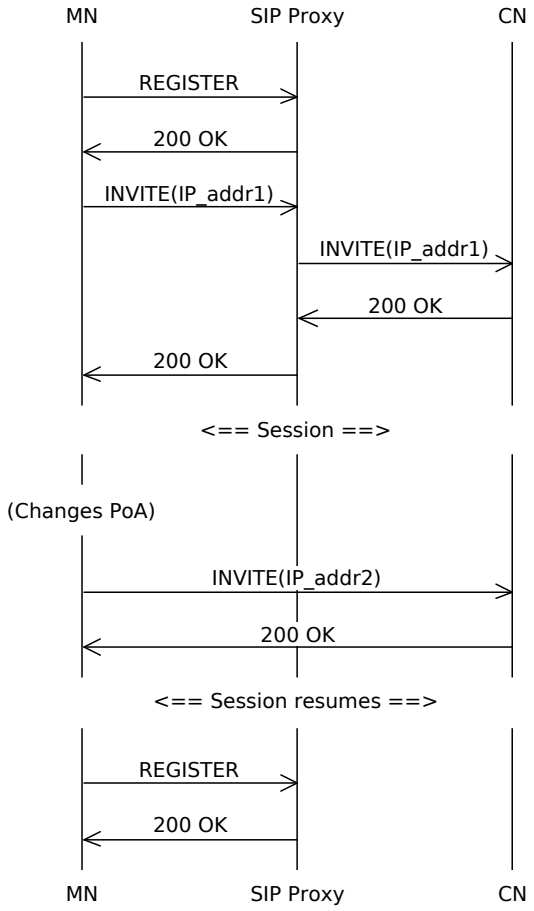


Figure 2.12: SIP: handover

	Mobile IP	**Proxy MIP**	**HIP**	**SIP**
Layer	3	3	3.5	5
Scope	Global	Local	Global	Global
Terminal support	Yes	No	Yes	Yes
Network support	Partial	Local domain	Partial	No

Table 2.3: IP based terminal mobility protocols

attachment without user intervention, while network mobility addresses the movement of intermediate nodes (routers).

Table 2.3 summarizes the properties of some IP-based terminal mobility solutions. The first row in the table indicates the layer of the protocol stack in which they operate. Mobile IP [42], for instance, is an IP (L3) mobility solution, works regardless of the link layer technology, and makes mobility transparent to TCP/UDP and applications. Host Identity Protocol (HIP), on the other hand, operates at layer 3.5, and so it is able to provide mobility support on top of either IPv4 or IPv6 transparently to TCP or UDP based applications. The Session Initiation Protocol (SIP) supports mobility at the "session layer" (L5) of the OSI stack. In general, a mobility solution requires that the location of the mobile terminal is tracked by an agent, and routes for packet delivery be rebuilt in real time.

Mobility solutions can also be classified by their geographical scope. Global mobility solutions, such as MIP or HIP, support any kind of mobility. Local mobility solutions, on the other hand, only support transparent mobility within a certain network domain, as is the case of Proxy MIP [43]. Access technologies, such as 802.16 or 802.11, support only local mobility. Global scope mobility protocols ensure that mobile nodes are always reachable, but they have poor performance because the mobility agent is located far from the moving terminals. Local scope mobility is more efficient but only achieves mobility transparency within a limited network region.

We may additionally characterize mobility solutions by the set of nodes requiring modifications. Solutions such as MIP or HIP require modifications in the terminal. Solutions like Proxy MIP are implemented by the network and require no modification in the terminals.

Network mobility solutions allow some intermediate nodes (routers) to change PoA without disrupting the ongoing user data sessions. In normal

IP networks, when an access router changes point of attachment it must change the IP address of one network interface. However, intra-domain routing protocols like Open Shortest Path First (OSPF) usually take a few seconds to detect topology changes and to install new routing tables in the remaining routers of the access network; during this time packets get lost. The NEMO protocol optimizes this scenario by extending the MIP approach to *mobile routers*. However, the NEMO protocol has problems dealing with multiple hierarchies of routers moving at the same time, creating nested tunnels.

As a conclusion we may state that there are multiple mobility solutions, operating at different layers of the protocol stack, with different scopes, and providing different transparency / performance trade-offs. In WNMTs, the need for a solution with a great degree of auto-configuration implies the use of ad-hoc networking. However, existing ad-hoc routing protocols were designed for networks with a few dozen nodes, and will not scale to thousands of nodes. In addition, their support for terminal or router mobility is inefficient, allowing several seconds to pass before node movement is properly detected by the network. Possible solutions for WNMTs, in addition to solving these problems, should incorporate mechanisms allowing the network to manage mobility on behalf of the terminals, as exemplified by Proxy MIP.

2.4 Network Simulators

Simulation is a method of studying a hypothetical system by creating a model of this system that behaves as much as possible as the real system but is easier to control. Experiments can then be carried out over the model; it can be replicated, stimulated, and measured, much easier than doing equivalent experiments using real equipment. For studying networking protocols, simulation enables the researcher to predict how a protocol will behave under conditions that are hard to replicate, for instance with hundreds or thousands of nodes, in a predictable way.

The most common technique for simulation of networking protocols is called *discrete event simulation* [44]. Discrete event simulation models a system by a series of discrete (but possibly infinite) states, and events that represent the transition between states. In these systems, the simulator has a notion of "simulation clock", which is a virtual clock to represent the hypothetical elapsed time in the simulated world. Events are *scheduled* to "fire" when the simulation clock has a certain value; when the event is fired, some programming instructions are executed that modify the model's state

in some way, and may schedule additional events. The main benefit of this technique is that the simulated clock does not have to be synchronized with the real world and can be advanced directly from one event time to the next event time, thereby simulating the system as fast as possible.

Although it is possible to write a simulator for any given system from scratch, following the general principles of discrete event simulation, it is not practical. Generic simulation frameworks exist that already do much of the work needed, and the programmer just needs to add the missing parts. This has two main advantages: (1) the existing simulator probably has the event scheduler much better optimized than one could write from scratch with limited time, (2) generic simulators often come with models for numerous systems that we need. The second point is particularly relevant for networking, as creating simulation models of protocols is hard, and often simulating a complete system requires models for many protocols besides the protocol being researched.

In this section, the some of the most relevant network simulation tools are introduced. Only a brief overview of these tools is given, with focus on aspects more relevant for Chapter 4, such as programming language used, number of available models, and license.

2.4.1 Network Simulator 2 (ns-2)

Network Simulator 2 (ns-2) has been traditionally one of the most widely used simulation tools in the networking research community: survey results for 111 published simulation papers in ACM's MobiHoc conference, 2000–2004, indicated ns-2 as the simulation tool used in 44.4% of papers [45], by a large margin the most used simulator. ns-2 is a discrete event simulator composed of a C++ part, containing *simulation models*, and a oTCL-based scripting interface. Most of the classes in the C++ part register themselves in the oTCL runtime, so that they become available to be used in oTCL scripts. In ns-2, the following object types are defined:

Application: an object that represents a process that generates packets to be transmitted;

Agent: the agent implements a specific transport layer, e.g. TCP;

Node: represents a typical network node;

Link: represents a link between nodes.

In ns-2, there is no concept of interface of a node, only links between nodes, which means the ns-2 structure makes it very difficult to simulate with detail systems with multiple interfaces. Nodes have IP and MAC addresses that are represented by simple integer counters. Although this is a simple scheme that allows some simple simulations to be written without concern for the actual addresses used, for more complex scenarios those actual addresses are of utmost importance and their simplification ends up being confusing.

Ns-2 Scripting

In ns-2, the scripting language used is OTcl: an object-oriented extension to TCL (Tool Command Language). TCL is a simple command-oriented programming language, with many similarities to Bourne Shell scripting syntax. TCL programs are essentially a concatenation of commands, and commands can have parameters separated from the command name by spaces. Variables can be defined, and later the value substituted by "dollar sign" syntax. All variable values are strings by default unless otherwise instructed to via an appropriate TCL operator. OTcl adds the keywords `Class` and `instproc` to allow one to declare classes and methods of those classes, respectively. While TCL is a widely used programming language, the oTCL extension is not widely used outside ns-2, making it more difficult to learn.

In ns-2, the "tclcl" (TCL with CLasses) library provides a layer of glue to allow exposing C++ classes to OTcl. Using this library to expose a C++ class to TCL involves making that class inherit from the tclcl class `TclObject`. We will follow the implementation of the class RTPAgent in ns-2 (`apps/rtp.{h,cc}`) as an example. The RTPAgent class is declared as follows:

```
class RTPAgent : public Agent {
 public:
        RTPAgent();
        virtual void timeout(int);
        virtual void recv(Packet* p, Handler*);
        virtual int command(int argc, const char*const* argv);
        void advanceby(int delta);
        virtual void sendmsg(int nbytes, const char *flags = 0);
 protected:
        virtual void sendpkt();
        virtual void makepkt(Packet*);
        void rate_change();
        virtual void start();
        virtual void stop();
        virtual void finish();
        RTPSession* session_;
        double lastpkttime_;
```

```
        int seqno_;
        int running_;
        int random_;
        int maxpkts_;
        double interval_;
        RTPTimer rtp_timer_;
};
```

As seen in the declaration, RTPAgent inherits from Agent, but inspection of the inheritance tree would reveal that Agent already inherits from TclObject, therefore RTPAgent also inherits TclObject functionality. Another step needed to expose the RTPAgent class to TCL is to declare a "class object", thus:

```
static class RTPAgentClass : public TclClass {
public:
        RTPAgentClass() : TclClass("Agent/RTP") {}
        TclObject* create(int, const char*const*) {
                return (new RTPAgent());
        }
} class_rtp_agent;
```

The above code causes the RTPAgent class to be registered as "Agent/RTP" in TCL; an instance of RTPAgent can therefore be created in TCL via the new operator (`new Agent/RTP`). Via TclObject, it is possible to "bind" C++ member variables to TCL, via bind() method calls in the constructor:

```
RTPAgent::RTPAgent() : Agent(PT_RTP), session_(0), lastpkttime_(-1e6),
    running_(0), rtp_timer_(this)
{
        bind("seqno_", &seqno_);
        bind_time("interval_", &interval_);
        bind("packetSize_", &size_);
        bind("maxpkts_", &maxpkts_);
        bind("random_", &random_);
}
```

For example, the following TCL code causes an RTPAgent object to be constructed and sets the member variable "seqno_" to be initialized to the value "123".

```
set rtp [new Agent/RTP]
$rtp_ set seqno_ 123
```

So far so good. Slightly more complicated is the binding of C++ methods to TCL. For this, the "command" virtual method of the TclObject class has to be overridden, and the new implementation must dispatch each possible supported "command" to the appropriate method call. The method receives an array of C strings; the first element of the array represents the command/method name, while the remaining elements are the command/method parameters, as shown in Listing 2.1.

Listing 2.1 RTPAgent::command listing

```
int RTPAgent::command(int argc, const char*const* argv)
{
        if (argc == 2) {
                if (strcmp(argv[1], "rate-change") == 0) {
                        rate_change();
                        return (TCL_OK);
                } else if (strcmp(argv[1], "start") == 0) {
                        start();
                        return (TCL_OK);
                } else if (strcmp(argv[1], "stop") == 0) {
                        stop();
                        return (TCL_OK);
                }
        } else if (argc == 3) {
                if (strcmp(argv[1], "session") == 0) {
                        session_ = (RTPSession*)TclObject::lookup(argv[2]);
                        return (TCL_OK);
                } else if (strcmp(argv[1], "advance") == 0) {
                        int newseq = atoi(argv[2]);
                        advanceby(newseq - seqno_);
                        return (TCL_OK);
                } else if (strcmp(argv[1], "advanceby") == 0) {
                        advanceby(atoi(argv[2]));
                        return (TCL_OK);
                }
        }
        return (Agent::command(argc, argv));
}
```

We have covered enough of the basics of ns-2 scripting to let us see some of the associated problems. The first problem is that the ns-2 TCL bindings are embedded into all ns-2 C++ classes, and cannot be removed or disabled. For the sake of reducing the memory footprint of the simulator, especially with thousands of nodes, and a few C++ objects attached to each node, it would be better if the scripting interface was an optional layer on top of a pure C++ core. Only if the scripting interface is needed should it be loaded into memory, and even then only objects exposed to the scripting interface should have a scripting state initialized. Another problem with the ns-2 scripting is that it is completely manual and, consequently, error prone. As an example, closer inspection of Listing 2.1 reveals expressions such as "int newseq = atoi(argv[2])" that convert a string to integer without error checking. We may consider as an additional shortcoming the fact that not all of the C++ API is exposed to TCL, only the methods that are explicitly supported by the developer, leading to the scripting interface often becoming like a "second-class citizen" which does not have access to everything. Finally, at the time of this writing, TCL is experiencing a decline in the mind-share of developers, as most new developers learn languages such as Python, Perl, Java, and C#, while TCL is becoming rather archaic. This was not a problem when ns-2 started—back then TCL was a very popular language—but is certainly becoming a problem now.

2.4.2 Network Simulator 3 (ns-3)

Network Simulator version 3, ns-3, is a new simulator that is intended to eventually replace the aging ns-2 simulator. ns-3 officially started around mid 2006, and the first stable version was released in June 2008, containing models for TCP/IP, WiFi, OLSR, CSMA (Ethernet), and point-to-point links, "God routing", among others. Additional stable versions have been subsequently released, including Python bindings, learning bridge, and real-time scheduler for version 3.2 (Sep. 2008), emulation, ICMP, and IPv6 addresses in ns-3.3 (Dec. 2008), WiFi improvements, object naming system, and "tap bridge" in ns-3.4 (Apr. 2009). Although ns-2 still has a greater number of models included in the distribution, ns-3 has a good development momentum and is believed to have a better core architecture, better suited to receive community contributions. Core architecture features such as a COM-like interface aggregation[46] and query model, automatic memory management, callback objects, and realistic packets, make for a healthier environment in which to develop new complex simulation models. In addition, it is reportedly [47] one of the better performing simulation tools

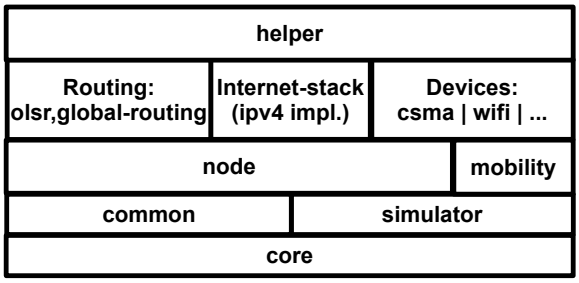


Figure 2.13: Overview or the main ns-3 modules

available today, is being more actively developed than ns-2, and has better code quality[48].

Modules

ns-3 is organized as a set of modules, as shown in Fig. 2.13. The "core" module provides additional C++ functionality to make programming easier, such as smart pointers [49], rich dynamic type system, COM-like [46] interface query system, callback objects, tracing, runtime described object attributes, among others. Other modules in ns-3 include "common", containing data types related to the manipulation of packets and headers, and the "simulator" module, containing time manipulation primitives and the event scheduler. The "node" module sits conceptually above the previous modules and provides many fundamental features in a network simulator, such as a Node class, an abstract base class for a layer-2 interface (NetDevice), several address types, including IPv4/6 and MAC-48 (EUI-48 in IEEE 802 terminology) address classes, and abstract base classes for a TCP/IP stack. The "mobility" module contains an abstract base class for mobility models. A MobilityModel object may be aggregated with a Node object to provide the node with the ability to know its own position. Certain NetDevice implementations, such as WiFiNetDevice, need to know the physical position of a node in order to calculate interference and attenuation values for each transmitted packet, while others, like PointToPoint, do not need this information at all. Some common mobility models are included in ns-3, such as *static*, *constant velocity*, *constant acceleration*, *random walk*, *random waypoint*, and *random direction* [50].

ns-3 also contains a couple of routing models, "olsr" and "global-routing", a module "internet-stack" implementing a UDP/TCP/IPv4 stack, and few NetDevice implementations, including WiFi (infrastructure and adhoc 802.11),

CSMA (Ethernet-like), and PointToPoint (very simple PPP-like link). Finally, sitting above all these modules is a "helper" module. This module provides a set of very simple C++ classes that do not use pointers (smart or otherwise) and wrap the existing lower level classes with a more programmer-friendly interface. In design pattern terminology [51], we might call this a *façade*.

Callbacks

In ns-3, a special Callback templated class is included, mainly to solve two problems. First, although in C++ it is possible to have callbacks to functions, via taking the address of the function using & and later using `(*callback)(parameters...)` to invoke the callback to function from its pointer. However, this approach does not allow one to take an address of a method of a specific C++ object instance. It is possible to take an address of a method in C++, but that address is not bound to a specific instance; to invoke a C++ method via method pointer an additional instance pointer is required and a special syntax is to be used. Ns-3 Callback objects allow both the method and instance to be stored in a single object, and allow the modules that receive and invoke the callback to use the regular calling syntax without having to worry about whether the callback refers to a function or an instance method. The other problem is related to a use case where some API needs to handle callbacks in abstract, regardless of the callback parameter types, and later invoke said callback with specific parameters. In plain C++, the API would have to take a "void *" as parameter, which is a generic pointer. Then, at invocation time the generic pointer callback is called with specific parameter types. Unfortunately, at the point in the program where the generic pointer is invoked with specific parameters the C++ compiler no longer has the information to verify that the receiving function/method callback can actually receive those parameters; type checking is disabled in this case. A programming error could easily lead to strange errors that are difficult to debug. The ns-3 Callback class also takes care of this problem by storing also the type of the callback function/method in the Callback object; this type is checked at invocation time (runtime), and an error message is given if the types do no match.

The following code listing illustrates how to create and use a callback object to a function. In this example, the callback object is declared as `Callback<double, int, float>`: the template parameters denote the return and parameter types of the callback. This callback object returns a value of type `double` and takes two parameters, of types `int` and `float`.

To bind a callback object to a specific function, the `MakeCallback` factory is used, taking as sole parameter the function identifier. Finally, we can observe that the callback object can be called as a normal function.

```
double MyFunc (int x, float y) {
   return double (x + y) / 2;
}

int main ()
{
  Callback<double, int, float> cb1;
  cb1 = MakeCallback (MyFunc);
  double result = cb1 (2, 3); // result receives 2.5
}
```

The following code listing illustrates how to create and use a callback object to a method of an object. The main difference in this case is in the way that MakeCallback is used, passing two parameters: the first parameter is the class method pointer, given by the C++ expression "`&MyClass::MyMethod`", and the second parameter is a pointer to the object instance. However, it should be noted that calling the callback object, `cb1`, is similar to the function callback case.

```
class MyClass
{
  public: double MyMethod (int x, float y) {
    return double (x + y) / 2;
  }
};

int main ()
{
  Callback<double, int, float> cb1;
  MyClass myobj;
  cb1 = MakeCallback (&MyClass::MyMethod, &myobj);
  double result = cb1 (2, 3); // result receives 2.5
}
```

Tracing

One of the unusual characteristics about ns-3 when compared to other network simulators is its tracing architecture. Generally, tracing is a facility provided by a simulator by which the user can discover which significant events are happening inside the simulation and under which conditions. Tracing will allow the researcher to derive important metrics of a simulation that can be used to quantify the value of a simulated model relative to another module. In ns-2, as in most simulators, tracing consists in generating a text file describing a series of events, with associated time stamps

and other properties, one event per line. Common events that are recorded to such files include MAC layer transmission, reception, and queue packet drop, among others. In ns-3, output of events to a text file is also provided, for a small subset of the events known by ns-3. Another possible way for storing events, in particular packet transmit/receive events, is via a PCAP files[7].

However, these are just alternative tracing systems; the main tracing system provided by ns-3 is callback based tracing. In ns-3 callback based tracing, *trace sources* are defined by ns-3 itself. Each possible trace source in ns-3 is associated with a specific object class and is identified by a name. The programmer may register a C++ function or method to be called when a certain (or a set of) trace source produces a new event. It is then the responsibility of the programmer to know what to do in the callback. Common uses for tracing include 1) writing raw event data to a file, 2) collect statistics for the occurrence of the event so that only the mean or other statistic moment is saved to a file, and 3) react to the event and change some parameter in the simulation in real time, for instance to experiment with cross-layer [52] optimizations.

Helpers

In order to ensure maximum modularity and extensibility, ns-3 models follow the approach of distributing the code over many small classes. Additionally, the classes are typically linked with (smart) pointers, instead of inheritance, following best "design patterns" programming practices, which states that one should favor composition, rather than inheritance, when creating associations between classes[51]. A unwanted side-effect of the composition and high number of classes is that this sort of API becomes complex for a researcher that wants to create a simple simulation program and is not interested in extending the simulation models. To overcome this issue, ns-3 employs another design pattern, called *façade*, which advises the software engineers to create an additional API that is simpler to use, and simply translates the simple API calls to lower-level object method calls. In ns-3, for each simulation model there is a set of low-level classes and also one or more *helper classes*. The helper classes are used by simulation programs, and direct access to low-level classes is discouraged. Although it is possible to write simulations entirely without using the helper classes, it makes the programming more complex and time consuming. Additionally, the API

[7]PCAP is a binary format for storing (usually live captured) packets, used by programs such as wireshark and tcpdump.

provided by the helper classes remains more stable across ns-3 versions. However, access to low-level objects is still provided for simulations requiring less common functionality[8].

2.4.3 OPNET Modeler

OPNET (Optimized Network Engineering Tools) Modeler is a widely used network simulator. It is a commercial product, although it has a free academic license. The main advantages of OPNET are its very complete graphical user interface, as well as reasonably complete set of models. On the other hand, its closed source nature makes it impossible to tailor the simulator to run in non-standard environments, such as embedded router platforms. Moreover, it's main programming language is C, thus programming new models is made more difficult than in more modern simulators due the complexity of memory management in C.

2.4.4 OMNET++

The OMNET++ [53] simulator was created to become an open-source alternative to OPNET. Although OMNET++ distinguishes itself from OPNET in being completely open source, its open-source license allows free use only in non-commercial settings. In OMNET++, scenarios are described in a new language called NED, forcing the developers to learn yet another programming language. Another drawback is that callbacks of scheduled events receive a single "message" parameter; all information has to be put inside this message, no more parameters may be added, which means more programming work. Additionally, it does not use reference counting or smart pointers, making memory management complex and error prone.

2.4.5 PARSEC, GloMoSim, QualNet

PARSEC [54] is a discrete event simulation framework that is based on C, developed by the Parallel Computing Laboratory at UCLA. In fact, PARSEC is a programming language that is an extension to the C programming language. The PARSEC compiler translates PARSEC programs into C programs, which can then be compiled into machine code and executed directly by the CPU. The basic building blocks of PARSEC simulator are:

[8] One might say that the motto "Simple things should be simple, complex things should be possible.", attributed to Alan Kay, applies in ns-3.

Entity: an Entity in PARSEC is like a function that can have its own flow of control. It works like a *process* or *thread* in a traditional operating system;

Message: a Message is a piece of data that can be exchanged between different entities. In fact, PARSEC's syntax for *Message* is very similar to C's `struct`;

Events: an event represents the passage of (simulated) time, and can be scheduled by a *hold(n)* PARSEC statement.

With PARSEC's new language constructs, which are enhancements to the C programming language, it becomes relatively intuitive to simulate physical systems based on discrete events. The PARSEC language model is not substantially different from Specification and Description Language (SDL) [55], a language that is widely used to model, simulate, and validate network protocols and state machines.

While PARSEC is a basic discrete event simulator, with no knowledge of nodes or protocols, GloMoSim is a network simulator built on top of PARSEC. It can simulate just a few link layers, such CSMA and 802.11, and just a few routing protocols, like AODV and DSR. The GloMoSim open source license allows it to be used for free in academic environments, and a commercial license can be purchased otherwise. It seems to be a discontinued product, no longer offering new releases for many years. Its successor, QualNet, has been rewritten in C++, no longer being based on PARSEC, and is a purely commercial product.

2.4.6 JiST / SWANS

JiST (Java in Simulation Time) [56] is a discrete event simulator written completely in Java, developed at Cornell University. It uses an unique approach of rewriting portions of the byte-code of normal Java applications in order to achieve event based simulation without requiring the developers to write code specific for simulation. In JiST, developers write classes normally, the only difference being that some classes have to be marked as *entities* which, similarly to PARSEC, represent objects that have independent / concurrent flow of control. The JiST compiler specifically rewrites `sleep()` calls to advance the simulation time, and rewrites method calls from one entity object to another to denote message passing and time synchronization. The main difference from, e.g., the ns-3 simulator, is that ns-3

is essentially a C++ library and the programmer needs to call ns-3 APIs, while JiST reuses pre-existing Java APIs.

On top of the JiST simulation kernel, the Scalable Wireless Ad hoc Network Simulator (SWANS) [57] was developed. It implements just a few routing protocols — AODV, DSR, and ZRP — and just 802.11 for link layer. On the plus side, any normal Java application can be run in the context of a SWANS simulated node. Nonetheless, clearly the SWANS simulator does not support many routing or link layer models, leaving it with a lot of potential but as yet unfulfilled. Perhaps the free-academic / paid dual licensing scheme inhibits greater model support from the community.

2.5 Conclusions

In this chapter we explained some technologies that are relevant for developing a network for metropolitan public transport systems where end user mobile terminals are strictly legacy terminals supporting WiFi and IPv4 with DHCP.

Regarding the link layer technologies, it is difficult to choose one in detriment of the others. WiFi is cheap, fast, and ubiquitous, but it has too short range to provide constant access to vehicles. WiMax has longer range, so it can be used to provide connectivity to buses; although costly to deploy, it may represent an opportunity for public transportation companies that wish to deploy its own networks. Otherwise, an UMTS network can be used to provide access to public transport vehicles. WiFi can be used inside buses and bus stops, to provide connectivity to end user terminals. WiFi, or the 802.11p variant, can also be useful to allow a bus to connect to a bus stop, or perhaps two buses communicating with each other. This will require intelligent routing, though. One final conclusion that we may draw from the link layer technologies is that in recent years the most important link layer technologies, with the notable exception of UMTS (and its evolutions), have been IEEE standardized and integrate well with IEEE 802 service model, for instance by using EUI-48 MAC unique addresses and integrating with the 802.1D Learning Bridge algorithm. Therefore, it makes sense to assume 48-bit MAC addresses and Ethernet-like frames as default, UMTS as the exception, and not the other way around.

When it comes to networking, there is a wide range of approaches possible. The 802.1D bridges use a very simple forwarding algorithm as long as the network is restricted to a tree topology, and STP can be run to ensure this type of topology. It does not require manual configuration, since it uses

only pre-configured (in hardware) addresses, but does not scale well because of STP and broadcast storms. The Internet Protocol, IP, scales very well, due to the use of structured addresses, but requires careful manual planning. Modifications to IP exist to handle adhoc networking, but the existing adhoc networking protocols, such as OLSR and AODV, do not scale to networks of hundreds of nodes. The MPLS approach is very interesting as data plane, as it is both fast and flexible, but it needs a good control plane that can handle dynamic network topology changes. TRILL is an interesting recently proposed network architecture, especially the proposed concept of a "virtual LAN" for terminals, and encapsulation of Ethernet frames over a network routed by a link state routing protocol adapter to work at L2.5, but it is not designed to handle mobility of nodes and does not solve the broadcast storm problem effectively.

Another topic considered is the existing approaches for supporting mobility of end user terminals. The main problem faced in IP networks is that a terminal has to obtain an address that structurally "belongs" to the router to which it is attached, meaning that if a terminal changes PoA it normally has to change IP address. The objective of a mobility protocol is to allow the terminal's communications to be minimally interrupted whenever it changes from one AR to another. This problem can be solved at any of multiple layers. For instance, SIP solves mobility at application layer by having the terminal send a new SIP INVITE message to the CN right after handover, to inform it of the IP address change, allowing the CN to start sending packets to the new address instead of the old one. The HIP approach is in a slightly lower layer, one could say layer 3.5; it defines a "host identity" layer, and allows end hosts to obtain cryptographic host identifiers that are logically above the IP addresses that the node may have on its interfaces. When a node's IP address changes, for instance handover to another AR (horizontal handover), or even handover to another technology (vertical handover), it notifies its peer correspondent nodes of this change via a HIP UPDATE message. The Mobile IP extension to IP works directly at layer 3 and requires no modification in applications, only in the IP layer. It allows nodes to have two addresses, one "home" address, denoting the identity of the node, and one "care-of" address, indicating the current location of the node. The mapping between home and care-of addresses is kept in a Home Agent node located in the terminal's home network. Correspondent nodes can send packets to the terminal's home address and the Home Agent takes care of tunneling the packet back to the terminals "care-of" address, this way ensuring that packets are not lost, even in the face of mobility. Fast Handover further optimizes this scenario by allowing the terminal to pre-

pare handover earlier, and tunneling packets from the old AR to the new AR for some time. The Proxy MIP approach is also interesting because it implements all the mobility signaling and tunneling work entirely within the network side, lifting the burden of implementing mobility from the terminal, and this way better supporting legacy terminals. Mobility solutions implemented at the link layer level, such as in UMTS networks, also allow the terminal to switch Point of Attachment (PoA) transparently to the IP layer, i.e. the IP address of terminal never changes.

Finally, some of the major network simulators were identified. The ns-2 simulator is still one of the most used simulators, but it has design problems such as lack of multiple interfaces support, obsoleted scripting programming language, lack of detail/realism, and difficult to extend data tracing subsystem. OPNET Modeler is also widely used, but it has restrictive commercial licensing, and the lack of source code makes it impossible to port to new platforms. OMNET++ tries to be an open source alternative to OPNET, but still the licensing is restrictive, allowing usage only in non-commercial settings. Additionally, the copyright assignment requirement for integrating code contributed by the academic community into the main code base is a major inhibitor for said contributions. PARSEC is a simulator that parses an SDL-like programming language, and translates it into C, for fast simulation. GloMoSim is built on PARSEC and adds some limited number of networking specific models, but it has been discontinued in favor of a commercial simulator rewrite, QualNet. JiST is a Java based simulation engine, while SWANS adds a few networking models. They too have restrictive licensing. Moreover, although fast, consumes substantially more memory than simulators written in C++. Finally, ns-3 is a rewrite of ns-2 from scratch, trying to solve many of the problems in ns-2. It has an open-source friendly license (GPL), is very efficient, and has a special focus on realism. Moreover, it has strong packet-level emulation abilities and real-time scheduling option, which can be exploited to make real-world experiments using the same protocol model developed for simulation; these abilities are further explored in Chapter 4.

Chapter 3

WiMetroNet

In this chapter, we propose an architecture for WiMetroNet that we believe is a good match for the WNMT introduced in Chapter 1. The proposed architecture works with the limitations of existing link layer technologies, be it range, cost, or capacity, and tries to combine them as much as possible. In addition, WiMetroNet is designed to scale to WNMT size, envisioning hundreds or thousands of moving routers inside the vehicles or in bus stops, and many thousands of end user mobile terminals.

3.1 Architecture

The WiMetroNet network, exemplified in Fig. 3.1, is generally structured in the following way. There are Rbridges in vehicles and bus stops or tram stations. They provide 802.11 connectivity to some vehicle equipments and to the users' terminals. Vehicles connect to the network core Rbridges through 802.16, while moving, or through 802.11 to the bus or tram stops Rbridges', while stationed near them; the Rbridges in bus stops or metro stations are connected to the core via high speed wired links, where possible, or fixed 802.16a wireless connections, for the most remote locations. At the WiMetroNet core a number of Rbridges are deployed in a Gigabit Ethernet mesh topology, and the WiMetroNet control plane ensures that optimum paths are used for forwarding traffic between different edge Rbridges. Finally, there is at least one IP router functioning as Internet gateway.

The terminals connect to one of the edge Rbridges and acquire an IP address through DHCP (the DHCP broadcast requests are tunneled to a well known DHCP server). The user traffic is then encapsulated when entering the WiMetroNet network, transported inside the network, and the original

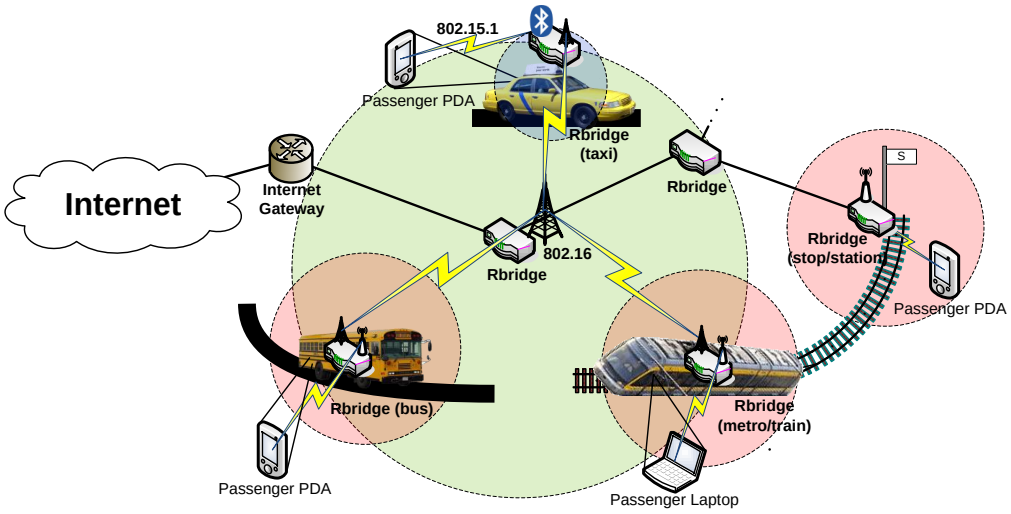


Figure 3.1: The WiMetroNet reference network

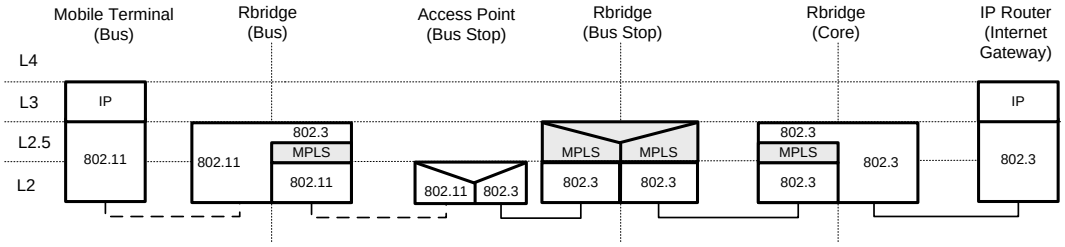


Figure 3.2: The WiMetroNet data plane stack

frames delivered to the destination station, or to the Internet gateway. Due to scalability concerns, in WiMetroNet broadcasts are strictly controlled: although single-hop[1] broadcasts work as expected, multi-hop broadcasts are forbidden by default, as recommended in [2]. Special algorithms and optimizations have to be employed for ARP, DHCP, and generic multi-hop service discovery. Unlike L3 adhoc networks, which use routing protocols such as OLSR and AODV, the WiMetroNet architecture does not require terminals to run any kind of special protocol, routing or otherwise; only the standard 802.11 family of protocols.

Because we want to support heterogeneous L2 technologies (typically, 802.11, 802.16, and 802.3), a new L2.5 header was introduced. Since the

[1]Up to the first Rbridge, covering e.g. a single bus.

Multi-Procol Label System (MPLS, RFC 3031) header is well known and fulfills our requirements, it was adopted for the WiMetroNet user plane. This MPLS header is very simple: it contains a 20-bit integer, for the "label", 3 bits marked "experimental" that are often used for QoS, one "bottom-of-stack" bit (for stacked MPLS headers), and an 8-bit TTL counter, totaling 32 bits. Using a standard MPLS header has the advantage of potentially allowing us to take advantage of MPLS switching hardware that already exists. On top of the MPLS layer the original L2 frames, from end terminals, are encapsulated, as shown in Fig. 3.2.

In the WiMetroNet architecture, heterogeneous technologies are supported, and the same Rbridge may have multiple interfaces, which are usually always active, though not all used. It is the routing protocol (WMRP) and its metrics that decides which interface to use at any given time for a given destination. In WiMetroNet, preference is given to 802.11 interfaces, then 802.16, and finally UMTS, although this ordering is configurable. When, for instance, a bus loses 802.16 connectivity, it will usually already have a UMTS link active, to be used as soon as the routing agent detects the 802.16 link failure.

The WiMetroNet routing protocol, WMRP, runs on Rbridges to disseminate topology information, allowing it to build MPLS paths. Also distributed by WMRP are the list of terminals (MAC identifiers) associated to each Rbridge, as well as the list of IP-MAC associations (DHCP leases). The MPLS based data plane works in coordination with the control plane in the following way. First, each Rbridge knows its own RID (Rbridge Identifier), which is unique in the network. By default (best-effort), packets are forwarded by applying an MPLS label that is the numerically equal to the RID of the egress (destination) Rbridge. Thus, no label negotiation protocol is required for the best-effort paths, only the routing protocol.

The greatest advantage of using MPLS as forwarding mechanism is that we open up the architecture to future extensions with no (or only minor) modifications to the data plane. For instance, we could add support for traffic engineered paths. We could begin by letting labels with values below $100,000$ to be reserved for RIDs, and the other values would be available for dynamic label assignments. Next, we would apply a traffic engineering protocol (e.g. RSVP-TE, adapted) to reserve a new path, by using only label values greater than $100,000$. From the point of view of the MPLS switching engine, there is no difference between best-effort path discovered by the routing protocol and engineered paths reserved by other means. Other examples of what can be accomplished using MPLS include Virtual LANs (VLANs), alternative backup paths for resilience to failure, and multicast

trees. Such improvements are possible without modifying or extending the encapsulation header used in the data plane.

In WiMetroNet, security in general is implemented using the traditional security mechanisms that exist in each L2 technology employed. For securing 802.11 links, the builtin security mechanisms can be used, such as a RADIUS[58] based authentication coupled with 802.1X[59]. It is also possible to integrate RADIUS based authentication with other technologies, such as 802.16. To protect the routing protocol messages, one simple solution is to apply a symmetric key cypher method, such as Advanced Encryption Standard (AES)[60]; the key has to be unique to each WiMetroNet instance, and shared among all Rbridges at deployment time.

3.2 Wireless Metropolitan Routing Protocol (WMRP)

The WiMetroNet control plane revolves around WMRP, a routing protocol that is inspired by OLSR, but which diverges from it in a number of ways. Like OLSR, WMRP is a link state, adhoc routing protocol designed for mobile wireless network. Like OLSR, WMRP defines HELLO messages for link sensing (discover neighbors), and TC (Topology Control) messages for disseminating network topology among all the nodes.

Among the differences between WMRP and OLSR we may include different address formats used, and new message types defined in WMRP to handle this type of network. While OLSR uses globally unique IPv4 addresses to identify each node participating in the adhoc cloud, in WMRP each node is assigned a unique 20-bit, MPLS compatible label. Additionally, WMRP defines two additional messages to cater for the needs specific to this type of network — MC (MAC Control) and IC (IP Control) — whose purpose is explained below. Finally, WMRP does not elect MPRs for use in flooding, for reasons stated at the end of this section.

3.2.1 WMRP PDU format

Like in OLSR, WMRP defines a *packet header* and a *message header*, depicted in Fig. 3.3. Each WMRP packet may contain a number of messages. The packet header contains packet size and a sequence number, while the message header contains fields such as *message type*, *validity time* (for how long is the information valid), *message size*, *time-to-live* (number of hops the message can still be forwarded before being dropped), *hop count* (number of hops it has been forwarded already), *originator id* (number of the node that generated the message originally). The *logical clock* field is used

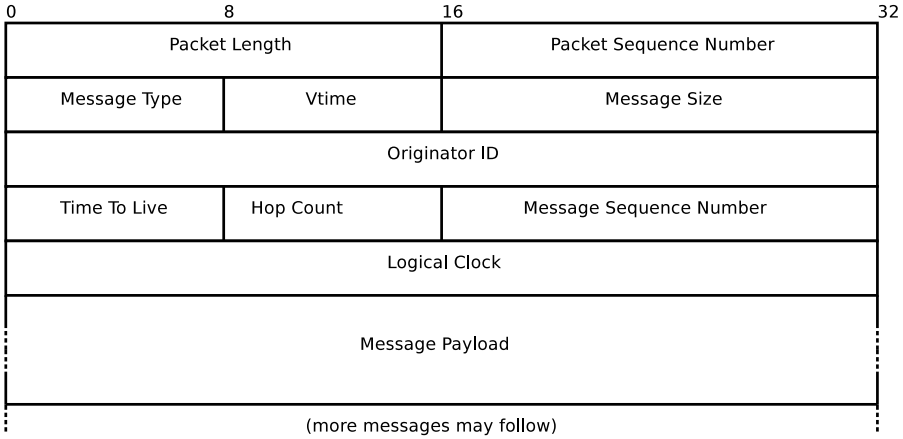


Figure 3.3: WMRP PDU format

for partial event ordering, and follows Leslie Lamport algorithm [61]. The payload of the message is to be interpreted according to the message type. Four message types are defined: HELLO, TC, MC, and IC.

The HELLO message is used for *link sensing*, i.e. to allow nodes to be discovered by their neighbors. HELLO messages are broadcast periodically (2 seconds, by default) by each node, but are never forwarded.

The Traffic Control (TC) message is used by each node to advertise to the rest of the network the list of links to neighbors it has discovered, along with "costs" associated with those links. The contents of a TC message is a vector of 32-bit fields; 20 of those bits represent the node id of a neighbor that has been found, while the remaining 12 bits store the link cost (or metric). The neighbor node IDs are discovered by listening to HELLO messages, while the link cost is a linear combination of factors such as bandwidth, delay, link usage monetary cost, and stability. The TC messages are generated periodically by each node and retransmitted by other nodes, after duplicates are eliminated, until the message reaches every Rbridge in the network.

The MAC Control (MC) message is similar in purpose to TC, but instead of advertising other WMRP-enabled nodes (Rbridges) it advertises a list of attached end-user terminals, each terminal represented by its MAC identifier (EUI-48). Like TC, MC messages are periodically generated and forwarded by all the other nodes.

The IP Control (IC) message is used to disseminate IP↔MAC associations. Typically, IC messages are generated only by Rbridges directly attached to a DHCP server, using the information contained in DHCP leases.

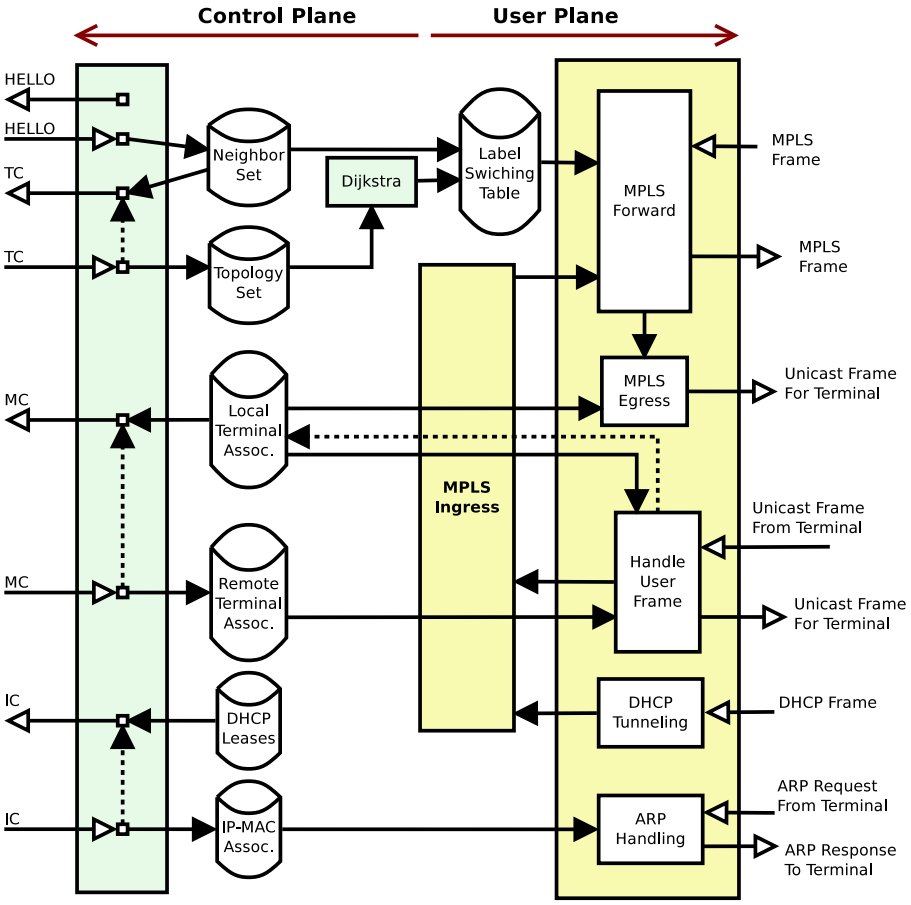


Figure 3.4: Interaction between control plane and user plane in a Rbridge

3.2.2 WiMetroNet Rbridge system architecture

We can see in Fig. 3.4 a schematic of the system architecture of a typical WiMetroNet Rbridge, including control plane, user plane, and the interactions between them.

Near the top-left of the diagram, the process of periodic generation of HELLO messages by the control plane is represented. The same control plane (i.e. the WMRP routing agent) may receive HELLO messages from immediate neighbors, and fill the *Neighbor Set* with a list of neighbors discovered so far. The Neighbor Set is consulted whenever a new TC message is periodically generated, while received TC messages feed information directly into a data structure called *Topology Set*. The Topology Set contains, for

each node participating in the adhoc network, a list of links, with cost, to its neighbors, thus forming a directed graph. This graph is fed into a Dijkstra Shortest Path algorithm, finally yielding information to be stored in a *Label Switching Table*. Periodically, MC messages are generated containing a list of attached terminals. The attached terminals are stored in a data structure called *Local Terminal Associations* (LTA), which can be filled by a method similar to 802.1D Learning Bridge, i.e. by inspecting the source address of incoming frames from terminals, or by L2 information, such as the list of associated 802.11 stations, in case of local 802.11 interfaces in AP mode. On the other hand, MC messages received from remote nodes will supply information to fill the data structure *Remote Terminal Associations* (RTA). Thus, by combining the information from Local and Remote Terminal Associations it is possible to find out, for each terminal MAC identifier, its current location, be it a local interface or a remote Rbridge. Some Rbridges may also periodically generate IC messages, using information supplied by a local DHCP server. Other Rbridges will receive the flooded IC messages and fill the *IP-MAC Associations* table.

The main user plane operations are also represented in Fig. 3.4, towards the right of the diagram. When an MPLS-encapsulated frame is received from a core-network interface, an MPLS forwarding operation takes place, using the information in the Label Switching Table. Then, the frame may either be retransmitted, remaining in the same MPLS tunnel (label switching), or it may be delivered to an attached terminal (egress). In the latter case, the Local Terminal Associations is also consulted to ascertain which of the local network interfaces the terminal is attached to. When a user data unicast frame from an attached terminal is received, the destination MAC is looked up in both Local and Remote Terminal Associations to determine how to handle the frame. If it is for a local terminal, the frame is simply retransmitted via the network interface that can reach the destination terminal. If, on the other hand, the destination terminal is instead found in the Remote Terminal Associations, the L2 frame enters the MPLS Ingress function, to be encapsulated and subsequently forwarded by the MPLS engine. Incoming DHCP frames are tunneled to a well known DHCP server, if the terminal is not yet know by the network, or the local Rbridge directly replies to DHCP requests, in case the terminal already known (from previous IC messages). Finally, ARP requests are intercepted by the Rbridge and an appropriate ARP reply is generated based on the information contained in the IP-MAC Associations table.

3.2.3 DHCP and terminal mobility

In the WiMetroNet architecture, all Rbridges are built to handle DHCP traffic by tunneling it to a well known DHCP server, and the reply tunneled back. End user mobile terminals are expected to acquire an IP address using this method, but what happens during handover (e.g. between a bus stop and a bus) is not so well defined. Since the SSID is the same in all buses and bus stops, the handover is a Layer 2 one from the point of view of the mobile terminals. In a L2 handover, the IP address stays the same, and the terminal does not need to renew the IP address using DHCP signaling. However, for increased robustness against misconfigured networks, some devices choose to use DHCP anyway, just to make sure they are still on the same network. In WiMetroNet, this is supported — the DHCP server will simply return the same IP address for a known terminal — although it makes the handover slightly slower[2]. In practice, *"Wireless IP Phone"* kind of terminals tend to favor L2 handover without DHCP signaling (for obvious reasons), while laptops tend to favor L2 handover with DHCP.

3.2.4 WiMetroNet: OLSR vs WMRP

Because OLSR is used as a basis for this work, we will try to highlight the differences between WMRP and OLSR by describing how OLSR would have to be used in the WiMetroNet scenario, and explaining why OLSR's MPR system is not used in WMRP.

OLSR applied to the WiMetroNet scenario

The main goal of the WiMetroNet scenario is to support legacy IEEE 802.11 terminals and manage mobility completely from the network side. To that end, an OLSR based solution for WiMetroNet would require OLSR to run only in the mesh routers (buses, bus stops, trams, base stations, core routers), and not in the end user mobile terminals. Because we do not want to require terminals to run Mobile IP, the IP address that terminals acquire through DHCP has to be stable, regardless of which Rbridge it is attached to in each moment. To keep the IP addresses stable, the Rbridge has to relay the DHCP requests from terminals to a well known DHCP server. OLSR's HNA (Host/Network Association) messages would be used to report to all the other Rbridges the current location of each terminal.

[2]Since the terminal is already known in the target Rbridge, the DHCP request is not tunneled, and the delay amounts to just one or two round-trips in the local 802.11 link.

To provide more insight into this solution, let's consider what would happen when a terminal hands off to a different Rbridge, e.g. from a bus stop to a bus:

1. The terminal loses WiFi connection to the bus stop and associates (L2 association) to the bus AP;
2. The terminal sends a DHCP request (broadcast);
3. The Rbridge connected to the bus AP intercepts the DHCP request and tunnels it to the central DHCP server;
4. The DHCP server notices a new request with a already known MAC address and sends a reply with the same IP;
5. The bus Rbridge receives the DHCP response from the DHCP relay tunnel, modifies the *gateway* address to its own address, and sends the reply to the terminal;
6. The terminal configures the IP address (a /32 address), which is the same as before, and configures the default gateway to be the bus Rbridge;
7. The bus Rbridge OLSR agent sends an HNA message once every 5 seconds, by default, to be flooded to the rest of the network. The next such message will include the IP address of newly attached terminal, causing other Rbridges to start forwarding packets for the terminal to this Rbridge.

As we can see, the OLSR based solution works at a different layer, and so the service offered to terminals is different. Depending on the operating system, it is likely that some application sockets are closed during handover, even though the IP address acquired in the new AP is the same as the previous one, because the handover is always a L3 one. In spite of the differences, it is easy to see that in the WiMetroNet architecture WMRP's MC messages play a role similar to OLSR's HNA messages. Thus, a WMRP protocol, configured with MC refresh interval equal to OLSR's HNA default refresh interval (5 seconds), can be considered to have approximately the same level of routing overhead and performance as OLSR. To be more precise, even under these conditions WMRP could be considered better than OLSR, since it may not drop application connections during handover, and does not need a DHCP signaling for each terminal handover, only for bootstrap.

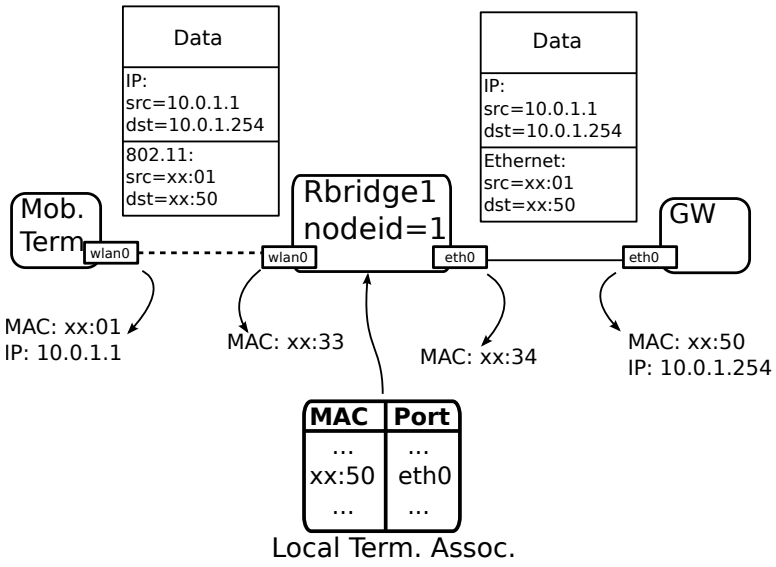


Figure 3.5: Two stations communicating over a single Rbridge

WMRP does not make use of the MPR optimization available in OLSR. Some of the reasons are given in Sec. 2.2.2. Additionally we should consider how MPR selection would impact the generation of MC messages. While in the case of TCs the MPRs take advantage of the fact that a link between two nodes is symmetric and can just as easily be advertised by either one of the nodes without loss of information, in the case of HNA messages, containing the list of terminals associated to each Rbridge, only that Rbridge has that information, since neighboring Rbridges may have outdated information. Therefore it would not be practical for an MPR Rbridge to generate an MC on behalf of another Rbridge, and therefore the MPR related savings do not apply in the case of MC messages.

3.3 WiMetroNet networking examples

In order to better understand how all the pieces of the WiMetroNet architecture fit, in this section some examples are given.

3.3.1 Two stations in the same Rbridge

The first analyzed scenario is a simple one; it consists of one mobile station, connected via 802.11 to an Rbridge that is linked to an Internet gateway

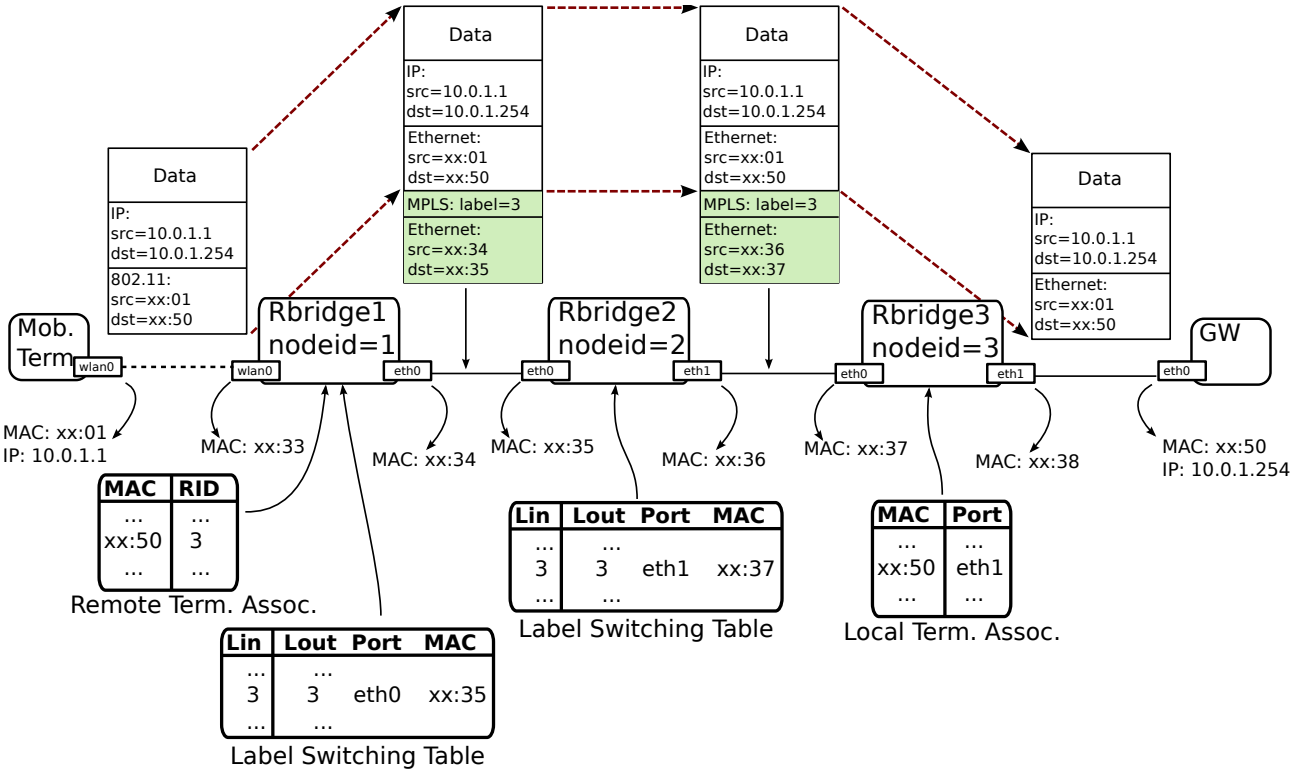


Figure 3.6: Two stations communicating over three Rbridges

via Ethernet. As shown in Fig. 3.5, the station sends a regular 802.11 frame towards the GW's MAC address, as if the Rbridge was a simple 802.11 AP. And, in fact, in this case the Rbridge behaves as an 802.11 AP: it simply forwards the frame to the ethernet port. Key to triggering this simple behavior is the realization by the Rbridge that the destination MAC address is listed in the Local Terminal Associations table, along with indication of the interface by which the destination MAC can be reached.

3.3.2 Two stations separated by three Rbridges

In this more complex scenario we exemplify what happens to a user frame that is encapsulated by WiMetroNet network. The frame undergoes three main operations: (1) ingress, (2) label switching, and (3) egress. This is shown in Fig. 3.6.

When Rbridge1 receives the 802.11 frame from the mobile terminal, it

looks at the destination MAC address, and discovers it listed in the Remote Terminal Associations. From there it discovers the destination MAC station is associated with Rbridge3, which has a corresponding nodeid 3, which is also the value of the MPLS label. The Label Switching Table indicates that the path (LSP) corresponding to label 3 continues with the same label value and the MPLS frame should be transmitted towards next hop with MAC address `xx:35` on interface `eth0`. Therefore, the original MAC frame is encapsulated in another frame containing two additional headers: 1) an MPLS header with label value "3", 2) a new "outer" Ethernet header, used just for the link between Rbridges 1 and 2.

When Rbridge2 receives the Ethernet frame, it discovers (from the "Ethertype") that it contains an MPLS packet. The MPLS header indicates a label with value "3". Since this label corresponds to a path to another Rbridge, the Label Switching Table is consulted; there it discovers that incoming packets with label 3 should be retransmitted with the same label value towards the next hop with MAC address `xx:37` via `eth1`. Thus, the packet remains the same, except that the TTL in the MPLS header is decremented.

When Rbridge3 receives the Ethernet frame, it discovers (from the "Ethertype") that it contains an MPLS packet. The MPLS header indicates a label with value "3". Since this label corresponds to Rbridge3's own identifier, the MPLS tunneling ends here and the outer Ethernet and MPLS headers are removed. The next step is to look at the inner Ethernet header, specifically the destination MAC address, `xx:50`. From Rbridge3's Local Terminal Associations table, it is discovered that the destination MAC station is connected to a local Ethernet port, eth1. Thus, the original Ethernet frame is transmitted by eth1, and reaches the gateway GW.

3.3.3 DHCP

Another illustrative example that is useful to present is the handling of DHCP signalling by WiMetroNet. As shown in Fig. 3.7, when an Rbridge receives a DHCP DISCOVER message, which comes with broadcast as destination IP and MAC addresses, the message is tunneled to the Rbridge which is directly connected to the DHCP server, in this case GW. Here we assume that Rbridge 1 is configured by the network operator with the MAC address of the DHCP server. Then, to discover the egress of Rbridge for the DHCP message, the RTA table is consulted, yielding an entry for that MAC address and Rbridge 2 as current location; this association was discovered by the routing protocol, WMRP. The DHCP OFFER follows a

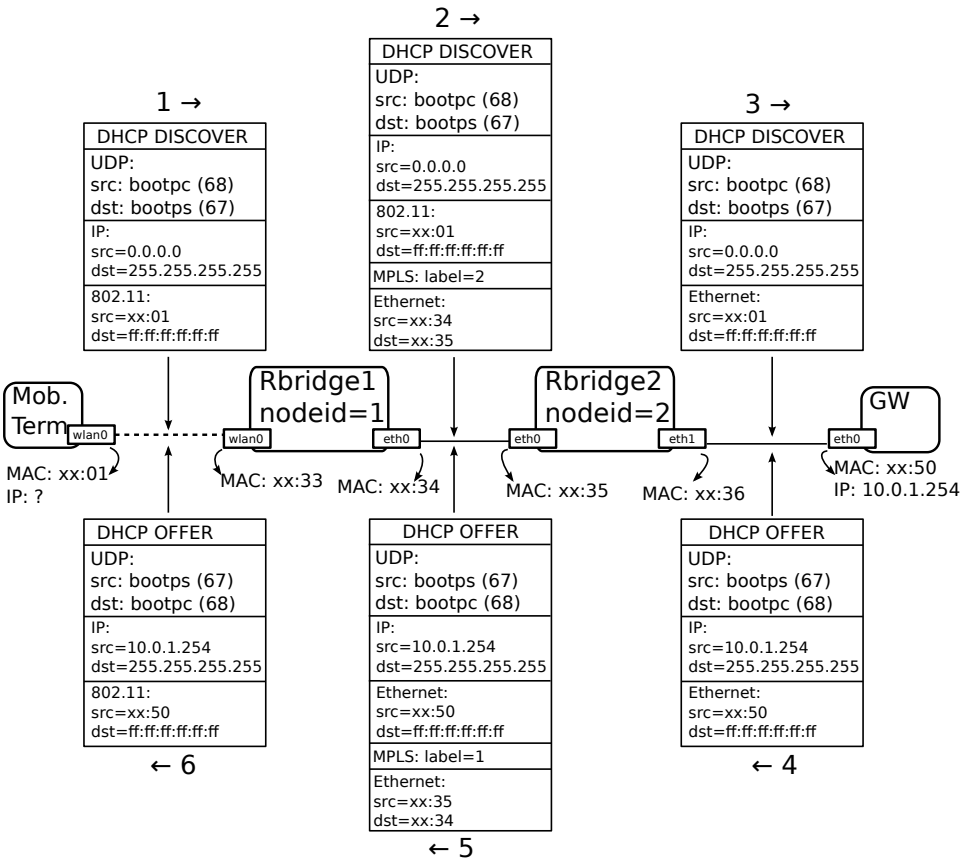


Figure 3.7: A mobile terminal acquiring IP address using DHCP

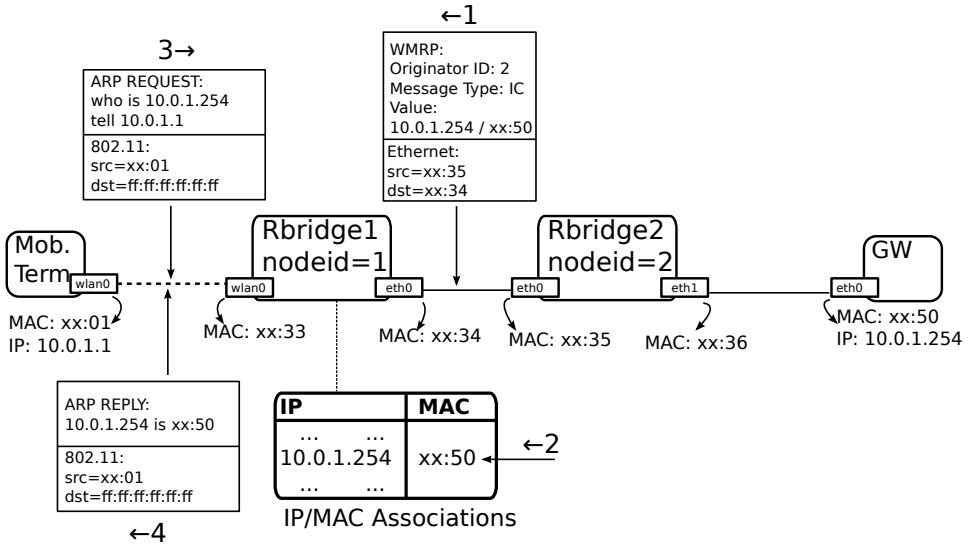


Figure 3.8: WiMetroNet ARP optimization example

reverse path towards the terminal. Similarly, DHCP REQUEST and DHCP OFFER messages are exchanged, but they are not included in the figure to keep it simple.

3.3.4 ARP

Fig. 3.8 illustrates how WMRP and the data plane interact to optimize ARP signalling from terminals. Once Rbridge 2 finds out about the IP/MAC association of one of the terminals[3], it uses the WMRP protocol to advertise this association to the other Rbridges in a WMRP "IC" message. Rbridge 1 receives this message, and adds the IP/MAC binding for GW into its own IP/MAC Associations table. Some time later, the mobile terminal may wish to send an IP packet to the GW, and so broadcasts an ARP packet requesting to find out the MAC address for IP 10.0.1.254. When this ARP request reaches Rbridge 1, it is not retransmitted; instead, it simply looks at the IP/MAC Associations, finds the MAC address xx:50 corresponding to IP 10.0.1.254, and forges an ARP reply to send to the mobile terminal. This optimization, besides avoiding flooding of the network with broadcasts, results in a shorter ARP request/reply round-trip, making the network appear more "responsive".

[3]For instance, by DHCP snooping

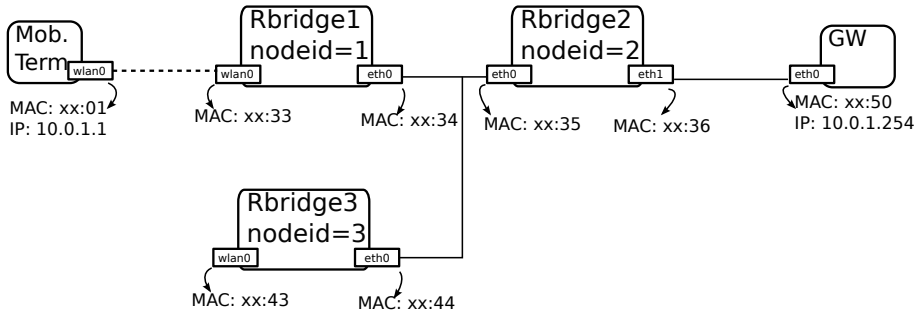


Figure 3.9: MC messages and terminal mobility: topology

3.3.5 WMRP: MC messages and terminal mobility

Next we consider a topology with three Rbridges and two terminals, shown in Fig. 3.9. A possible exchange of packets is shown in Fig. 3.10. Let us assume that, a time $t = 5s$, the mobile terminal (mt) sends a MAC frame to rb1 (the destination address is not important). When rb1 notices the source MAC address of the MAC frame, it assumes a terminal with that MAC address is attached to the wlan port, wlan0. Thus, it adds the terminal MAC address to its LTA. Assuming the refresh interval for MC messages is 5 seconds, at time $t = 5$ the Rbridge rb1 broadcasts a new MC message with the contents of LTA. This way, rb2 is informed that the terminal with MAC address xx:01 (mt) is currently located at an Rbridge with ID 1 (rb1), and stores this information in its RTA. Later on, if a station gw, attached to rb2, sends a MAC frame towards mt, rb2 looks at the destination MAC address, consults its RTA, and discovers that the destination MAC address is for a terminal located at remote Rbridge with ID 1 (rb1). Thus, it encapsulates the frame and sends it towards rb1, where the encapsulation is removed and the original frame transmitted to mt.

We may additionally examine in detail how the network handles handover of a terminal. Suppose the mobile terminal mt detaches from rb1 and then attaches to rb3, in other words a handover, as shown in Fig. 3.11. In this case, rb3 will soon detect mt, either via an incoming data frame or via 802.11 infrastructure mode association events, and will add the mt MAC address to its LTA. When the next MC is scheduled to be transmitted, it will contain this new mt MAC address. The Rbridge rb2 will thus discover that the mt is now at Rbridge 3 instead of 1. The next frame from gw destined to mt will then be tunneled to rb3 instead of rb1.

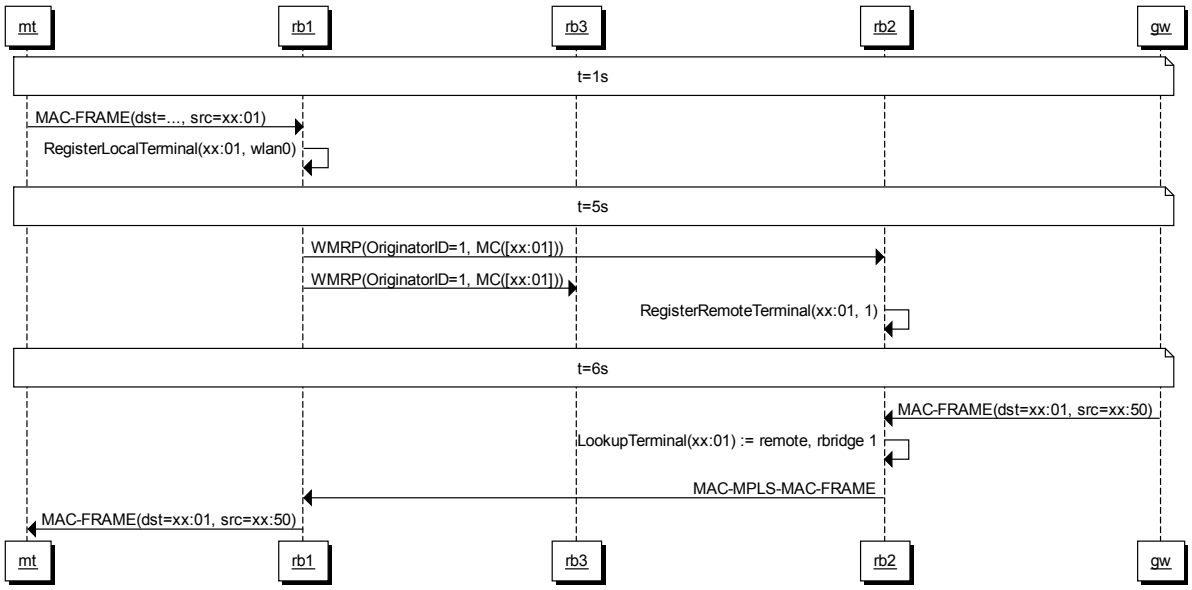


Figure 3.10: MC messages and terminal mobility: MSC

3.4 WiMetroNet software architecture

While trying to evaluate the WiMetroNet architecture, we had to make some ns-3 contributions, some of which are detailed in Chapter 4, and develop a simulation model for WiMetroNet itself. The same simulation model is being used as basis for a prototype deployment, using ns-3 techniques described in Chap. 4, hence the more generic title "software architecture" being used in this section.

3.4.1 Overview

The WiMetroNet software module is organized as three main components, as shown in Fig. 3.12. A separate instance of each of these components is installed on each ns-3 Node, i.e. hundreds/thousands in the case of simulation, or a single Node in case of a implementation deployment. The basic functionality of each component can be summarized as follows:

MPLS contains an MPLS forwarding engine, forming the core of the WiMetroNet-the data plane. It uses the ns-3 Node and NetDevice APIs to register protocol handlers[4] and transmit MPLS Ethernet frames;

[4]In ns-3, a "protocol handler" is a callback that is registered to handle frames with a

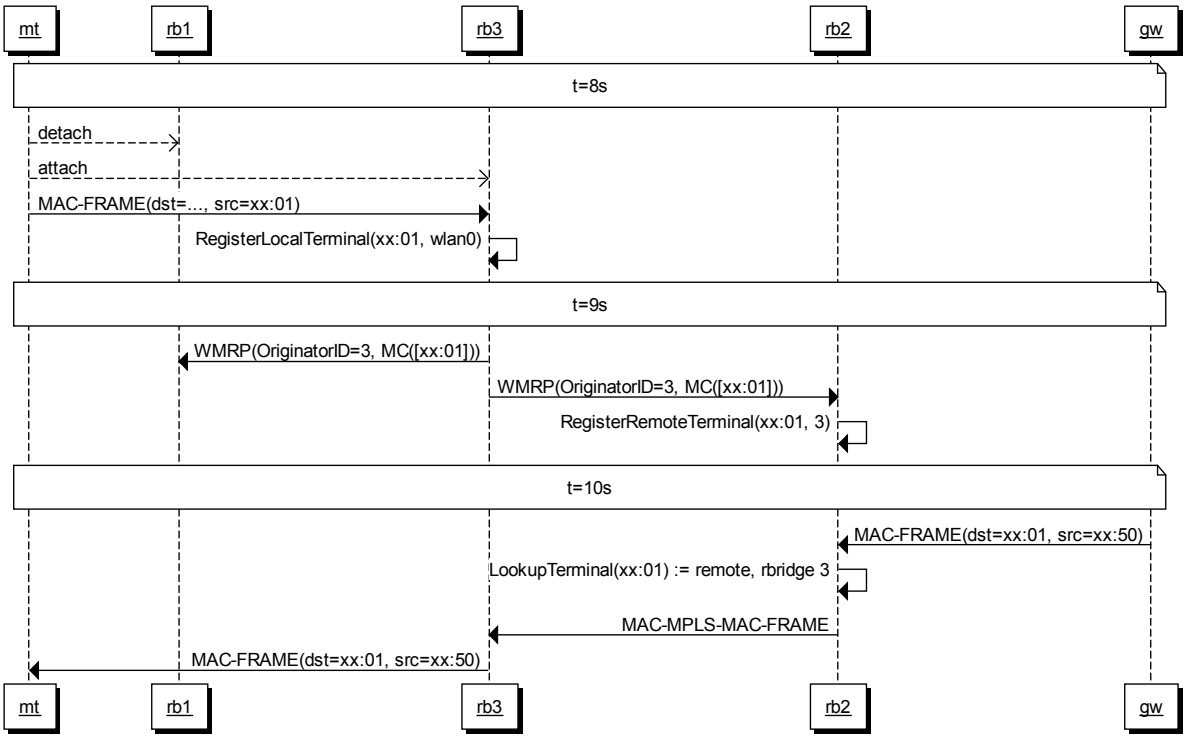


Figure 3.11: MC messages and terminal mobility: handover MSC

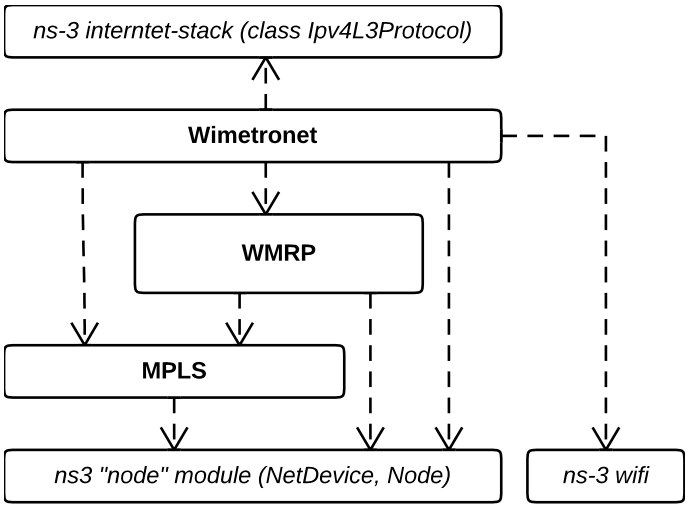


Figure 3.12: High-level view of the wimetronet software architecture

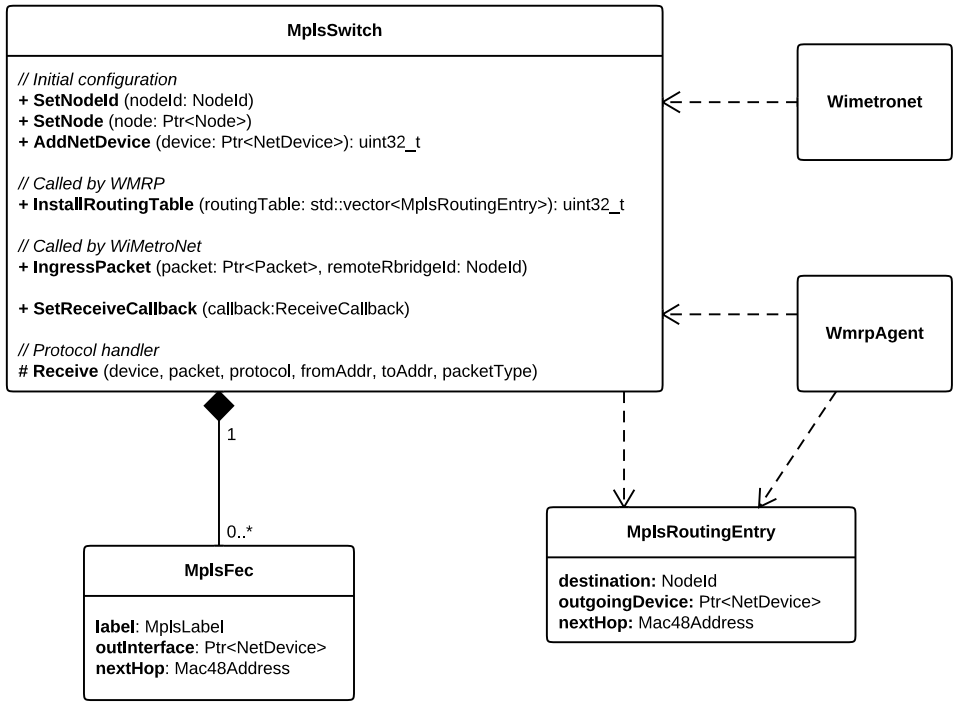


Figure 3.13: WiMetroNet software architecture: MPLS class diagram

WMRP contains the WMRP routing agent, with abstract input/output interfaces;

Wimetronet contains the "glue" code to orchestrate the other two components. It instantiates the routing agent and handles input/output of WMRP PDUs. It detects local terminal associations[5] and reports them to the routing agent, and handles ARP requests from terminals. Finally, it delivers to the Node's IPv4 stack packets that are received from MPLS and are clearly destined to that Node.

3.4.2 MPLS

Fig. 3.13 contains a UML class diagram for the MPLS module. The main class to be found is MplsSwitch, which is basically an MPLS forwarding

certain protocol type.

[5]To detect associated terminals, WiFi AP-mode MAC layer association notifications are used.

engine. Its main methods are:

SetNodeId: this method configures the "NodeId" that represents the identity of the Rbridge where the MplsSwitch is installed. The MplsSwitch uses this ID to compare against the labels of incoming MPLS frames, and deliver to upper layers in the stack frames which are destined to this Rbridge and so encapsulation must occur;

SetNode: stores a pointer to the Node object, so that the AddNetDevice() method can register a protocol handler;

AddNetDevice: this method asks the MPLS switch to monitor the given interface for incoming MPLS frames.

InstallRoutingTable: this method is called by the WMRP routing agent whenever new routes are discovered. The sole parameter is a list of *routing entries*, represented by the MplsRoutingEntry structure, including the following fields:

destination: the NodeId of a destination Rbridge to be reached;

outgoingDevice: the interface of the Node from which the *destination* Rbridge can be reached;

nextHop: the MAC identifier of the "next hop" Rbridge;

Clearly, this is a routing table suited to MPLS; the WMRP agent only conveys the routing information needed by MPLS, and not all of the information it possesses. The MPLS switch converts each MplsRoutingEntry into an equivalent MplsFec structure, which has basically the same information, but is conceptually independent;

IngressPacket: this method can be used to request the MPLS switch to "ingress" a packet, i.e. request that a packet enter the MPLS L2.5 network to be delivered at the remote Rbridge, given by the *remoteRbridgeId* parameter;

SetReceiveCallback: configures a "callback" to be invoked when a packet is received in MPLS and which has "egressed", i.e. a packet whose MPLS label matches the NodeId of the current node;

Receive: internal method used as "protocol handler" for MPLS frames. This method removes the MPLS label and either forwards the packet to another node by inserting a new label (label switching), or delivers to the upper layers by calling the *receive callback*.

3.4.3 WMRP

The WMRP is a component that contains the class `WmrpAgent` (WMRP routing agent) and associated data structures, such as `LocalTerminalAssociations` and `RemoteTerminalAssociations`, as shown in Fig. 3.14.

The class `LocalTerminalAssociations` implements the functionality already described Sec. 3.2.2 (see Fig. 3.4 in page 62), i.e. it holds the set of user terminals associated to the local Rbridge. The associations are represented by a associative array (`std::map`), with the MAC identifier of the terminal as key, and a `PortAssociation` structure as value. The `PortAssociation` structure contains two fields: 1. the interface of the Rbridge to which the terminal appears to be associated, and 2. the expiration time of association, allowing terminal associations to be expired if not periodically refreshed. The class `Wimetronet` is responsible for adding or refreshing terminal associations, while `WmrpAgent` reads those associations and advertises them to the network via MC WMRP messages.

The class `RemoteTerminalAssociations` holds the set of terminals association associated to remote Rbridges. The associations are represented by a associative array (`std::map`), with the MAC identifier of the terminal as key, and a `RemoteAssociation` structure as value. The `RemoteAssociation` structure contains two fields: 1. the ID Rbridge to which the terminal appears to be associated, and 2. the expiration time of association, allowing terminal associations to be expired if not periodically refreshed. The class `WmrpAgent` is responsible for adding or refreshing terminal associations, thanks to reception of MC WMRP messages, while `Wimetronet` reads those associations to handle ingress of MAC frames into the MPLS domain.

The class `IpMacAssociations` holds a set of (IP, MAC) pairs discovered via WMRP, namely IC messages. Conceptually, it represents the set of *IP leases* in registered in the Rbridge to which the DHCP server is attached.

The main class, `WmrpAgent`, implements the WMRP routing agent, and has one-to-one associations with `LocalTerminalAssociations`, `RemoteTerminalAssociations`, `IpMacAssociations`, and `MplsSwitch`. Its main public methods are:

WmrpAgent: this is the main contructor, and it requires as parameters the pointers to the above mentioned objects, plus the NodeId;

Start: this method is invoked, after everything is configured, to start the agent;

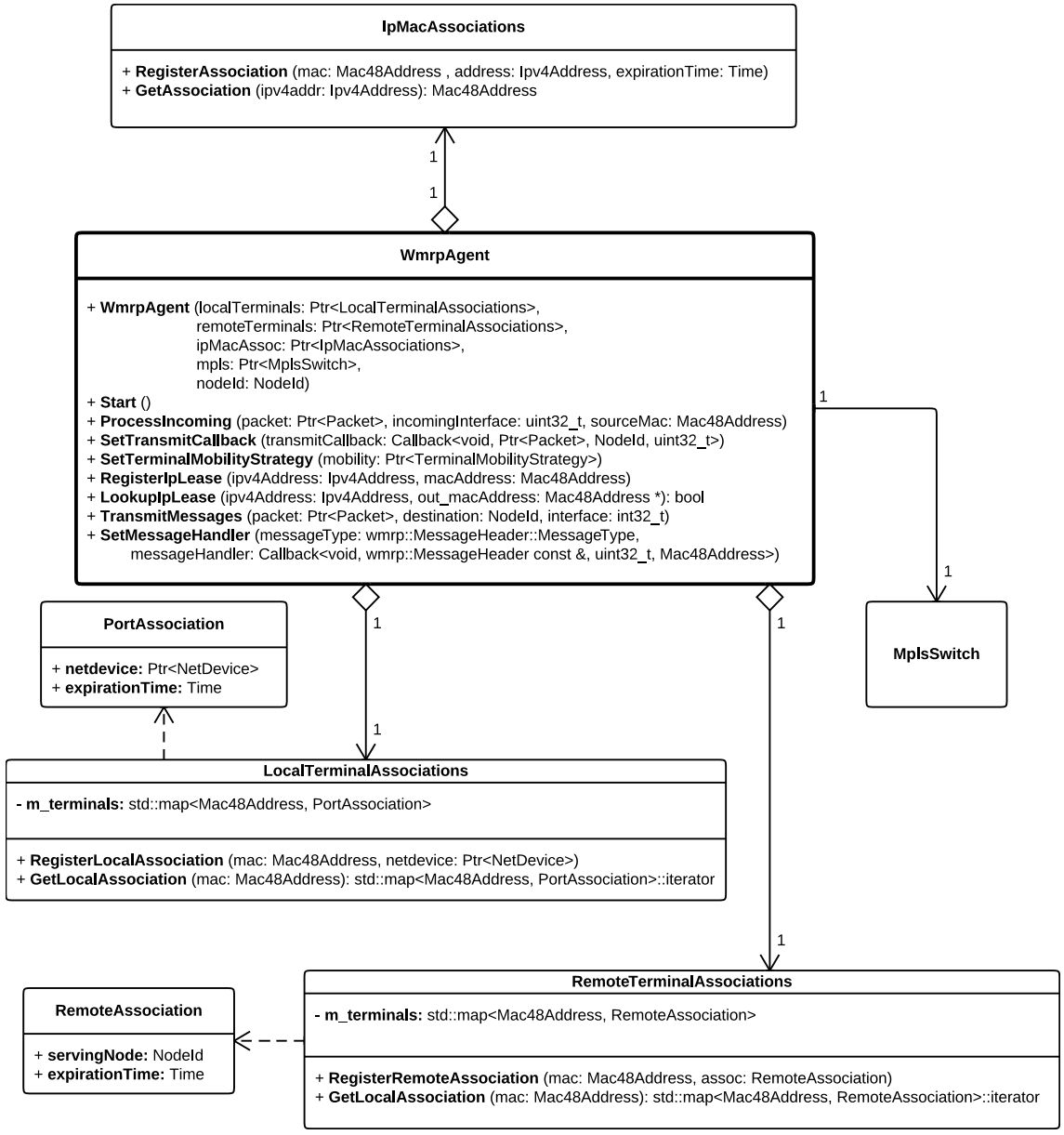


Figure 3.14: WiMetroNet software architecture: WmrpAgent class diagram

ProcessIncoming: when the Wimetronet class intercepts a message of the WMRP protocol type, it forwards the packet to the WMRP agent by calling this method with the full packet, interface, and source MAC identifier;

SetTransmitCallback: with this method we can register a callback to be invoked by the WMRP agent whenever it has a new PDU to transmit. The callback will receive as parameters the PDU payload, and number of the interface on which the packet should be transmitted;

RegisterIpLease: method that registers an IP lease, i.e. IP-MAC association. It is called directly by the simulation script, since there is no DHCP simulation model in ns-3, and simulating the full DHCP protocol is not important for the results we wanted to obtain.

LookupIpLease: this method looks up, in the local IP leases database, the IP address that is registered for a terminal. It is used by the Wimetronet class, in conjunction with the `IpMacAssociations` object, to generate a reply to an ARP request;

TransmitMessages: this allows code external to the WMRP agent to inject WMRP messages to be transmitted as if they were coming from WmrpAgent itself. This functionality is used by the `TerminalMobilityStrategy` class, to be described later;

SetMessageHandler: this method is also meant to be used by `TerminalMobilityStrategy` to extend WMRP functionality. It basically registers a callback that becomes responsible for handling a certain WMRP message type.

3.4.4 The "Wimetronet" component

What we call "Wimetronet" component encompasses the `Wimetronet` class, to realize the WMRP agent input/output interfaces, MPLS ingress/egress operations, and ARP optimizations. Additionally, the class `TerminalMobilityStrategy` defines a framework for terminal mobility optimization "plug-ins", with two implementations defined for the optimizations described in Sec. 3.5. To make this section easier to follow, detailed description of the `TerminalMobilityStrategy` class is postponed to Sec. 3.4.6.

The main methods to be found in class `Wimetronet` are:

Wimetronet: the contructor takes several parameters, which are also stored as member pointers of the class:

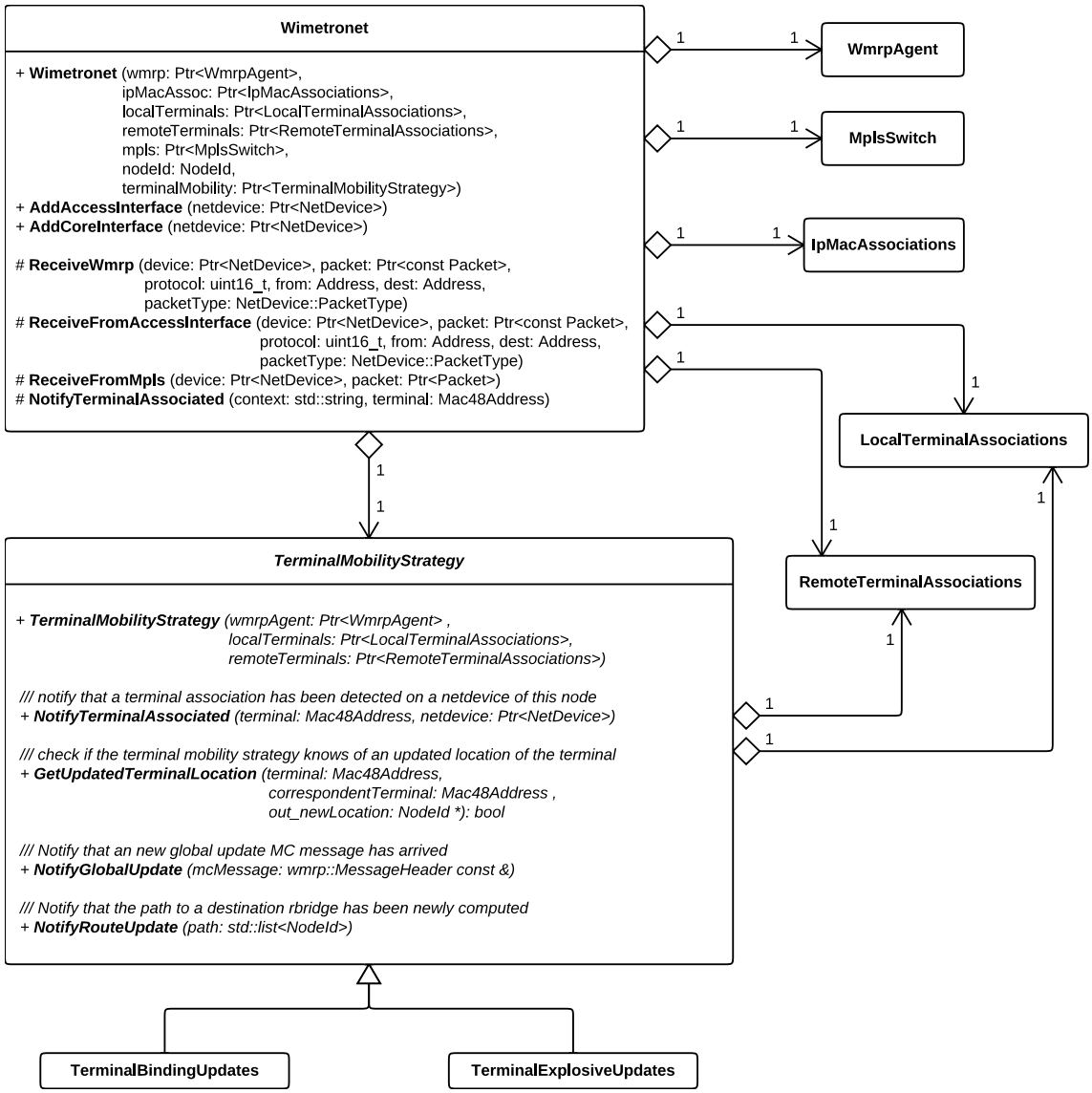


Figure 3.15: Wimetronet and TerminalMobilityStrategy class diagram

1. the WmrpAgent, to call its ProcessIncoming method when a new WMRP PDU is received, and transmit outgoing PDUs on its behalf,
2. the IpMacAssociations object, used to implement the ARP optimization,
3. the LocalTerminalAssociations object, to be able to handle frames for local terminals, either egressing from MPLS, or received from other terminals in the same Rbridge,
4. the RemoteTerminalAssociations, to be able to handle MPLS ingress of terminal frames destined to another terminal in a remote Rbridge;
5. the MplsSwitch object, which is called to ingress frames for remote terminals, and register a callback for frames leaving the MPLS tunnel,
6. the NodeId of this Rbridge (used just for debugging log messages);
7. the TerminalMobilityStrategy object which, as mentioned previously, is used to optimize terminal mobility;

AddAccessInterface: this method is called by the simulation script to declare that an interface of the Rbridge is an "access interface", meaning that only end user terminal data frames are expected on that interface and no WMRP or MPLS frames are processed or even expected. In addition to registering a regular "protocol handler", in order to receive the user terminal frames, this method specially handles WiFi AP interfaces by registering "MAC station association" events, to allow for faster detection of associated terminals;

AddCoreInterface: the counterpart of AddAccessInterface, but for "core interfaces", i.e. interfaces of an Rbridge that connect to other Rbridges. This method calls MplsSwitch::AddInterface, to let the MPLS switch receive MPLS frames, and registers a protocol handler for WMRP PDUs;

ReceiveWmrp: this method is the actual handler for WMRP protocol that is registered by AddCoreInterface. It simply calls the method ProcessIncoming of WmrpAgent with the packet payload and sender MAC address;

TransmitWmrp: this method is registered as "send callback" with WmrpAgent. It takes care of actually transmitting WMRP PDUs on behalf of the WMRP agent;

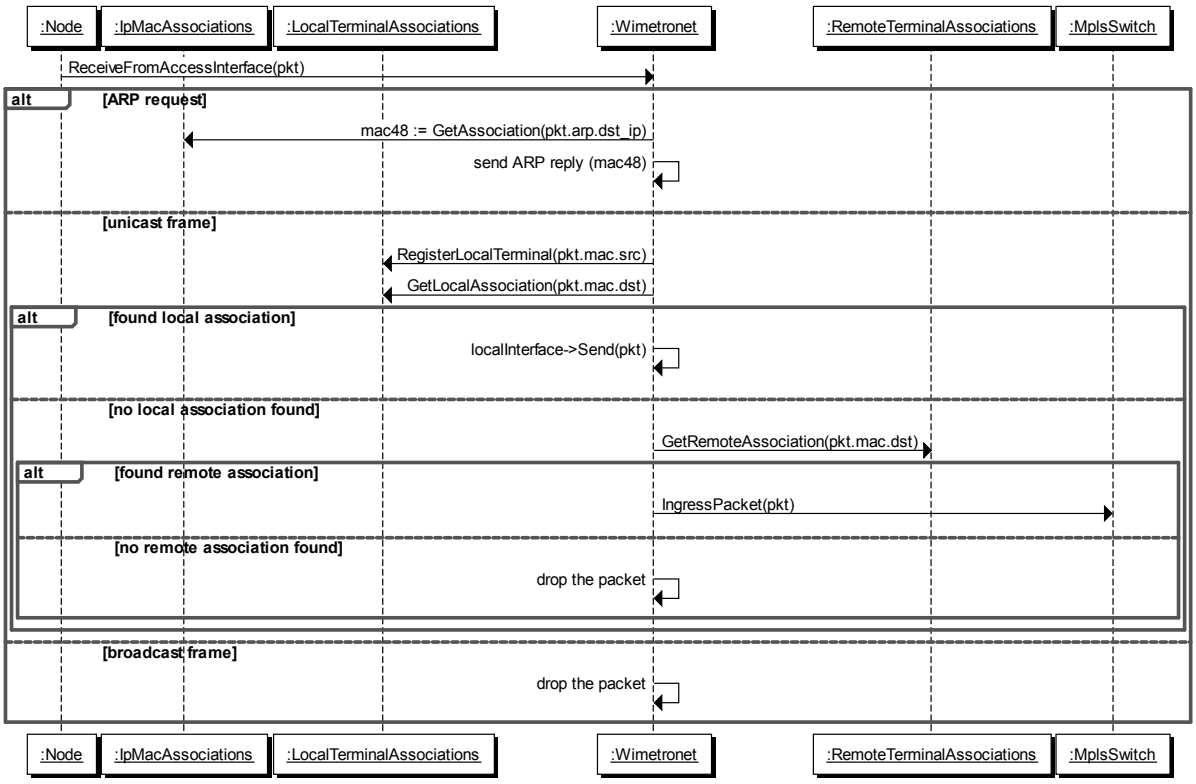


Figure 3.16: Wimetronet::ReceiveFromAccessInterface

ReceiveFromAccessInterface: in this method, Ethernet frames from end user terminals are processed;

ReceiveFromMpls: this method is called by MplsSwitch whenever an MPLS frame arrives destined to this Rbridge;

NotifyTerminalAssociated: this is the method that is called to handle association events for WiFi interfaces of the Rbridge. It registers the terminal association in the LocalTerminalAssociations object.

3.4.5 Sequence diagrams

Wimetronet::ReceiveFromAccessInterface

The sequence diagram in Fig. 3.16 illustrates what may result of calling the ReceiveFromAccessInterface method the Wimetronet class, to process a

user MAC frame. This method checks with local and remote terminal associations to find out where is the terminal with destination MAC address is located. If it is in a local Rbridge port, the frame is simply retransmitted on that port. If it is on a remote Rbridge, the frame is encapsulated in MPLS and delivered to the MPLS layer. One additional important task of this method is to notify the LocalTerminalAssociations that the terminal has been "seen" in that local bridge port. This way, even if the interface is not WiFi, and so does not have an explicit association event that may be monitored, the location of the terminal is still discovered implicitly from the traffic it generates. Finally, this method also checks for ARP request broadcast frames, and generates corresponding ARP replies using information obtained from the IpMacAssociations object. This sequence is illustrated by

MplsSwitch::Receive

Fig. 3.17 shows the method Receive of MplsSwitch being called whenever an MPLS frame arrives. The method first removes the MPLS header and looks at the label. If the label does not match the ID of the Rbridge then it means it must be forwarded by MPLS. The MPLS forwarding consists in looking up the FEC for this label, in the Label Switching Table, adding a new MPLS header with the new label, and transmitting the packet through the network interface indicated in the FEC.

In case the label matches the ID of the Rbridge, the packet is given to the method ReceiveFromMpls of the Wimetronet class, for further processing. Here the destination MAC address of the packet is checked to see if it matches one of the local interfaces, in which case the packet is assumed to be meant for the local Rbridge stack itself; Ipv4L3Protocol::ForwardUp is invoked, this way delivering the packet to the local IP stack. If the destination MAC address is not of a local interface, the packet is assumed to be destined to a user terminal. The MAC address is searched in local and remote terminal associations. If local, the packet is simply transmitted to the terminal. If remote, the packet ingresses again into MPLS for remote delivery.

Sending a WMRP PDU

The sequence for sending a WMRP PDU, implemented by Wimetronet-::TransmitWmrp, is shown in Fig. 3.18. This method takes as parameters the WMRP payload to transmit, a destination Node ID, and an interface. When

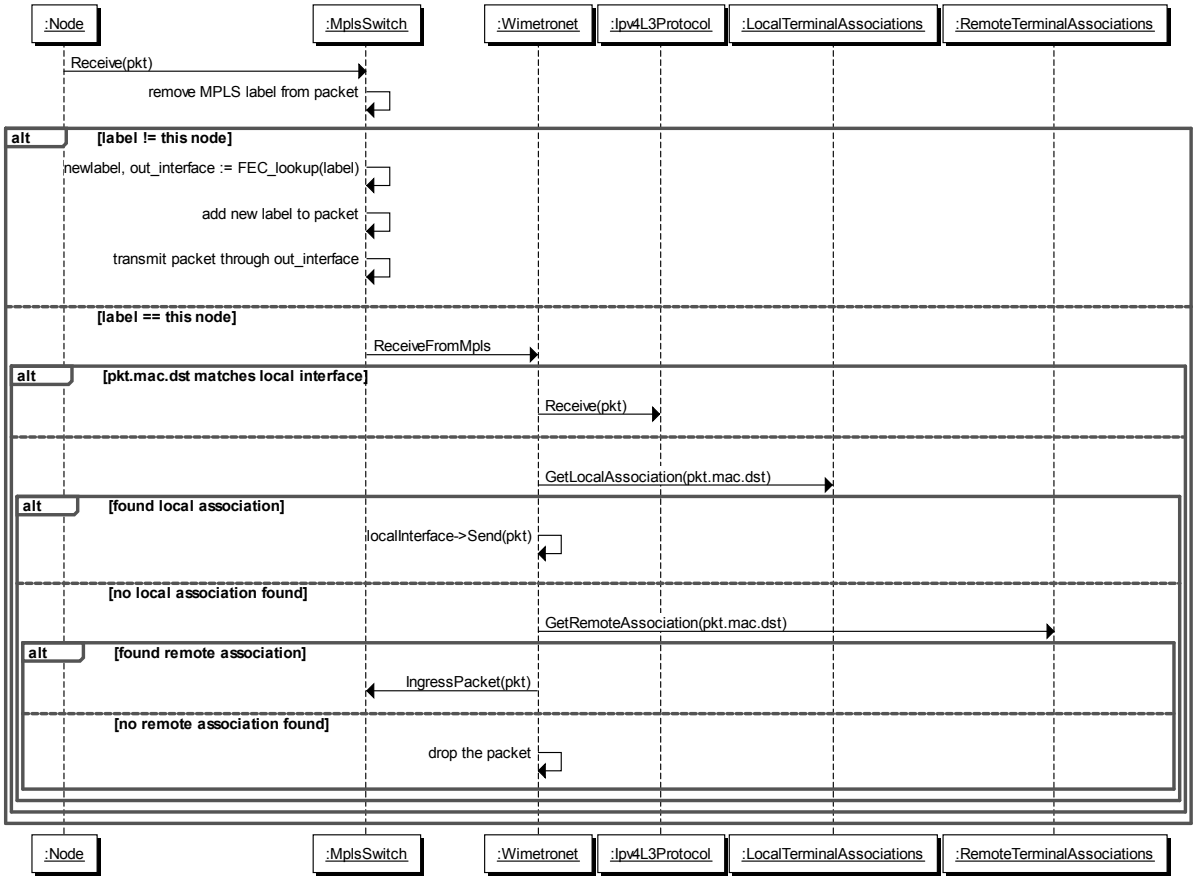


Figure 3.17: MplsSwitch::Receive sequence diagram

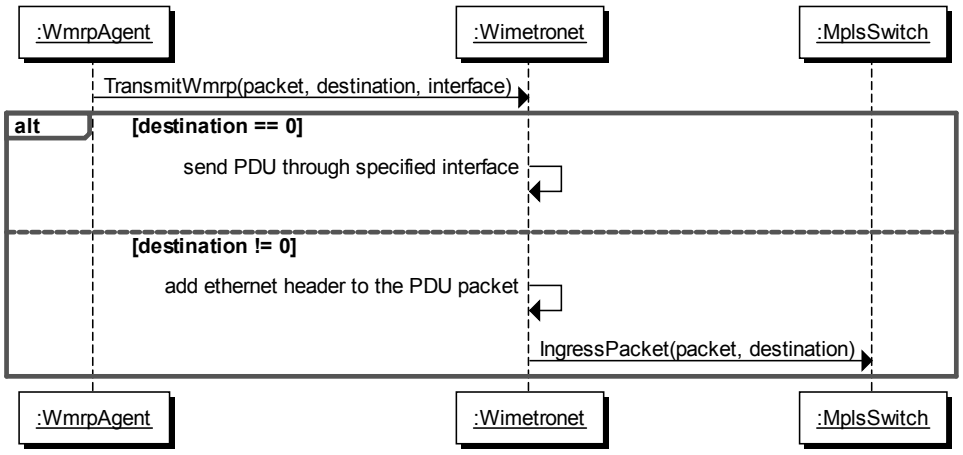


Figure 3.18: Sequence diagram of sending a WMRP PDU

the "destination" parameter is zero, the WMRP PDU is simply transmitted through the indicated interface. If the destination Node ID is nonzero, it means the caller wishes to indicate a specific destination Rbridge to which this PDU should be delivered; this is implemented by adding an Ethernet header to the WMRP PDU and ingressing the packet to MPLS for remote delivery.

Receiving a WMRP PDU

Receiving WMRP PDUs is shown in Fig. 3.19. There are two possibilities for WMRP to be received: from MPLS or from an interface directly. When a WMRP is received via an MPLS tunnel, the MplsSwitch object processes the packet normally and delivers it as usual to the Wimetronet object via the ReceiveFromMpls method. Here, the WMRP EtherType is recognized, and so the payload is delivered to the ProcessIncoming method of WmrpAgent. If, on the other hand, the WMRP PDU is received directly from a network interface, it is recognized by its EtherType and also delivered to WmrpAgent::ProcessIncoming.

3.4.6 The TerminalMobilityStrategy class

With the WiMetroNet simulation framework, it has always been our objective to be able to experiment with multiple terminal mobility optimization strategies, or even disable such optimizations completely for the sake of

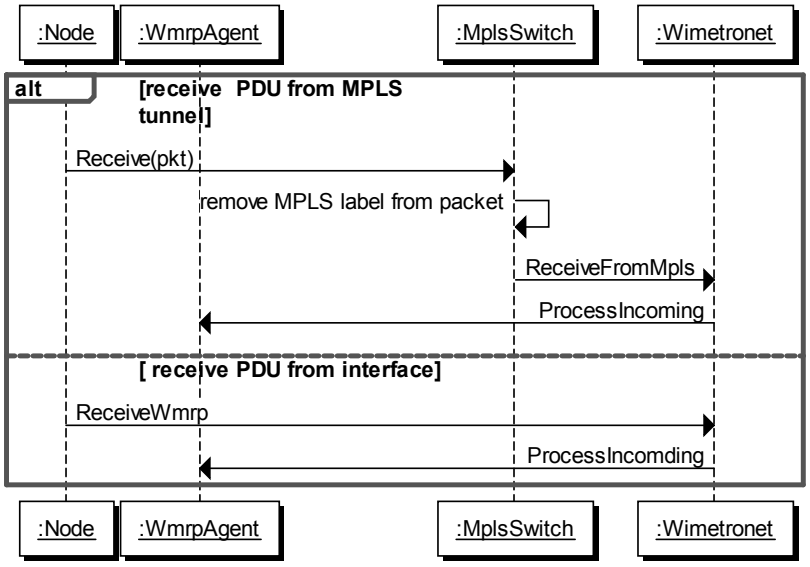


Figure 3.19: Sequence diagram of receiving a WMRP PDU

baseline comparison. However, programming all the options directly into the main simulation model would lead to many conditional branches and make the system more difficult to write and maintain.

To avoid this problem, a system for extending the base WiMetroNet functionality in a modular fashion was devised. This extension system was inspired partly in two well known approaches. One of these approaches is *aspect-oriented programming* (AOP) [62], which allows one to structure a system as a core part plus additional *"aspects"*. Each "aspect" provides additional functionality, which can be "weaved" into the core. In AOP implementations, the core and aspects are separate entities, whose source code is kept separate, and the weaving is done by a compiler. This way, both core and aspects become smaller and easier to understand. However, since a full AOP system, with respective compiler, seemed excessive overhead just for this sake, we looked for an alternative that would employ only standard C++ Object Oriented programming. The adopted solution was based on the "Strategy" design pattern [51], and is realized by the *TerminalMobilityStrategy* class, represented in Fig. 3.15.

The class defines an interface which subclasses can implement in order to "advise" the `Wimetronet` class of updated terminal location. This is done in the following way. First, the Wimetronet class keeps the `Terminal-`

`MobilityStrategy` informed of relevant terminal mobility events by calling the methods `NotifyTerminalAssociated`, `NotifyGlobalUpdate`, and `NotifyRouteUpdate`. Then, the `Wimetronet` class will call the method `GetUpdatedTerminalLocation` of the strategy class, which allows the strategy to inform `Wimetronet` whenever it has found a more up-to-date terminal location. This is done whenever a terminal location lookup is required, namely in ingress and egress operations, right before looking up the terminal location in `LocalTerminalAssociations` and `RemoteTerminalAssociations`. Besides the `Notify*` method calls, the `TerminalMobilityStrategy` implementations are allowed to access the `WmrpAgent` to send and receive additional WMRP PDUs, using the `SetMessageHandler` and `TransmitMessages` methods of `WmrpAgent`.

3.5 Fast and scalable terminal mobility

What has been described so far is the base WMRP architecture, which is a simple link state routing protocol. In this base architecture, if we copy the OLSR default settings, WMRP periodically broadcasts HELLOs every 2 seconds, and periodically generates a new TC and a new MC every 5 seconds. As for the IC messages, they may be generated with a frequency as low as once every 60 seconds, since they convey information (DHCP leases) that changes very slowly.

In link state routing protocols, such as OLSR and WMRP, the interval between periodic routing messages that are to be flooded through the entire network, such as TC and MC, is a crucial design setting, a trade-off between routing control traffic overhead and convergence time with mobility. Although the 5-second interval is not a bad choice for supporting mobility, it can be easily shown that it does not scale very well to large networks. The rate P of MC messages received by a node is given by $P = \frac{N-1}{\tau}$, with N representing the number of Rbridges in the network τ the message generation interval. As the routing control traffic increases linearly (on a per link basis) with the network size, there is a limit to the network size that is reached when the amount of control traffic exceeds a reasonable fraction of the network capacity.

To overcome these limitations, we begin by lowering the rate of control traffic to one message per 60 seconds. This reduces the control traffic to one twelfth of the normal value, but leaves the routing protocol unable to adapt to node mobility in a timely manner. On top of the periodic global routing messages, additional control messages are defined and used, in order

to support mobility, effectively but without flooding the entire network, thus scaling better with the network size.

In the following sections two methods of supporting mobility of end user terminals in an efficient and scalable way are described. Then, simulation results are presented and discussed. Only the routing overhead due to terminal mobility is addressed. While there are mobile Rbridges in the simulations (inside buses, which are actually moving), the routing overhead caused by them is neither optimized nor considered in this thesis. To account for the routing overhead of moving Rbridges, which generate TC messages, we reserve 5% of link bandwidth; together with the 5% limit we consider for the MCs (terminal mobility), we limit the total routing overhead to 10% of the link bandwidth, therefore leaving at least 90% of the network capacity for transporting user traffic. These values are just examples of limits that seem reasonable. If we determine that a network can scale to N nodes with 10% of routing overhead, we can expect that the network can scale to M nodes, $M > N$, if the routing overhead limit can increase to e.g. 15%.

3.5.1 The "explosive updates" approach

The solution that we call "explosive updates" is based on the technique of generating an MC message with a limited TTL to notify a small region of nodes around the former point of attachment of the terminal whenever a handover occurs. We call this an "explosive update". Due to this local update mechanism, a distant correspondent Rbridge computes a path to another Rbridge that ends just inside the edge of the explosive update region around the destination Rbridge. The strategy consists in reaching the closest node that was for sure updated if the terminal moved since the last global link-state update. Packets will follow this path/tunnel, egress at the region edge, find a new more up-to-date route, and ingress again, this time on the right path. To obtain the RID of the Rbridge at the edge of the explosive update region, we take advantage of the fact that the Dijkstra Shortest Path algorithm outputs a full shortest path to the destination Rbridge, not just the next hop, in other words a list of RIDs. Determining the egress point for the packets is then a matter finding the n^{th} RID in the list counting from the end, n being the explosive update TTL that is configured.

Consider as an example the scenario in Fig. 3.20, wherein a mobile terminal T changes point of attachment from node D to node F. The network topology consists of a grid of wireless nodes, with each node having as neighbors the nodes immediately above, below, left, right, and diagonals. A corresponding node (another terminal) is attached to node A and is sending

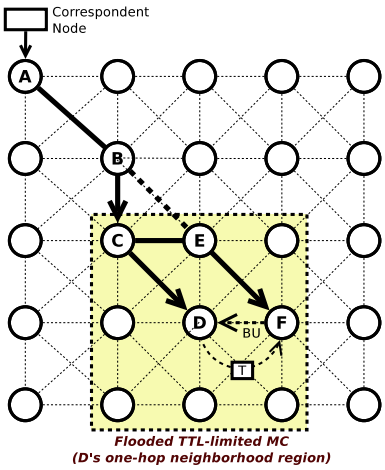


Figure 3.20: The "explosive updates" terminal mobility solution

packets to our terminal of interest T. Let us consider, for demonstration purposes, that the explosive update is limited to TTL=1. This TTL value is just an example and is configurable.

Initially, packets take the shortest path, which is split into two contiguous MPLS tunnels, A, B, C followed by C, D. Thus, two ingress operations occur in this case, one in node A and another in node C. Node A sends the packets to egress on node C because it knows that C is the node in the shortest path from A to D that is closest to A but still within the one-hop region around D, thus guaranteed to remain informed of the latest location of the terminal. As packets arrive at C, they egress the first MPLS tunnel and ingress into the second one, C, D, finally arriving at the desired destination.

When the handover of the mobile terminal from D to F occurs, a BU (Binding Update) message is sent from F to D, which simply notifies D that the terminal, T, is in a new location, F. Consequently, D emits a new MC message with TTL=1, this way advertising the new location of the terminal to its one-hop neighborhood, which includes C and E. Meanwhile, A is unaware of the topology change, and keeps sending packets to node C. However, since C has been notified of the handover, it starts forwarding the packets to the correct new location. Thus, the second MPLS tunnel is no longer C, D, it has now become C, E, F.

With this solution, we have frequent updates due to mobility, but they are localized and therefore scale well for large networks. Moreover, the solution is still purely proactive, and the overhead is always the same regardless

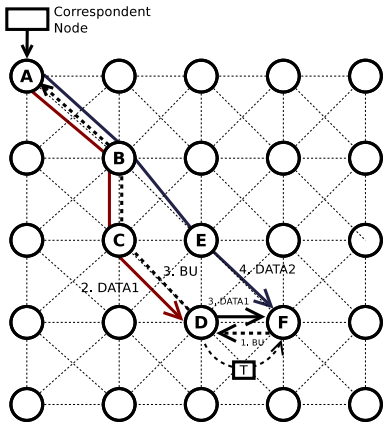


Figure 3.21: The "binding updates" terminal mobility solution

of the pattern of traffic and number of correspondent nodes. There are some drawbacks too. Even in static situations, two MPLS tunnels are used most of the time (except for nodes that are close to each other), leading to nearly twice the rate of ingress operations, which is a more expensive operation than MPLS forwarding. Additionally, during handover the routes become slightly less optimal. In the example, after the handover the optimal route would be A, B, E, F, but the actual route traversed by packets is A, B, C, E, F, which has one more hop and, consequently, larger delay. Unless the terminal moves again, this effect will last for 30 seconds on average, until the next global MC update notifies also A that the terminal has moved.

3.5.2 The "binding updates" approach

This mobility optimization consists of notifying the Rbridge attached to the correspondent nodes, reactively, when they send packets to the old location of a node that has already moved. As an example, consider the scenario in Fig. 3.21. As soon as the terminal handover from node D to F occurs, a BU message is transmitted from F to D to notify it of the handover. However, node A is not notified, and keeps sending packets for the terminal towards node D. The first packet after the handover that D receives, DATA1, is forwarded by D to the correct new node that is currently serving as the terminal's point of attachment, F. Additionally, a BU message is sent by node D to node A to notify it of the new terminal location. Thus, future packets sent by A will, from then on, take the correct shortest path between

A and F: A, B, E, F.

The "binding updates" solution is a hybrid proactive / reactive approach: it is proactive due to the slow periodic global MC updates, but is also reactive due to the BUs sent to correspondent nodes as data packets arrive at the incorrect locations. The main advantage of this solution is that packets always follow the shortest path, even after the handover, with the exception of the first packet[6] of the flow, which takes the wrong path and therefore experiences a slightly larger delay. The overhead of this approach is minimal for common scenarios. It depends mainly on the number of correspondent nodes, or, to be more precise, the number of different Rbridges attached to correspondent nodes. This works very well for common scenarios of server-client communication, especially if all the servers are behind the same Rbridge (e.g. hosts on the Internet). However, for peer-to-peer applications, with a large number of peers evenly distributed among many Rbridges, the overhead generated by the binding updates is expected to become significant.

3.6 Evaluation

In this section we evaluate the WMRP routing protocol, with focus on the terminal mobility optimizations that were described. For evaluation purposes, we consider that the base WMRP protocol with MC interval of 5 seconds to be approximately equivalent to the OLSR protocol. We consider two scenarios: a "road" scenario, representing a single bus line, and a "grid" scenario that mimics a city grid with city blocks and buses traveling between them.

3.6.1 Road scenario

Scenario

To evaluate our routing protocol, we started by a simple scenario that consists of providing coverage to a single straight bus line, with regular bus stops. This "road scenario" is not only simple enough to allow us to derive analytical expressions for the routing overhead, but also offers some realism.

It is depicted in Fig. 3.22, where we can see it mainly consists of a straight road, with a number (β) of equally spaced bus stops, a number ($\frac{\beta}{2}$) of equally spaced base stations, and a pyramidal topology of additional infrastructure

[6] Or rather, the first few packets, depending on the sending rate and round-trip time between ingress and egress nodes.

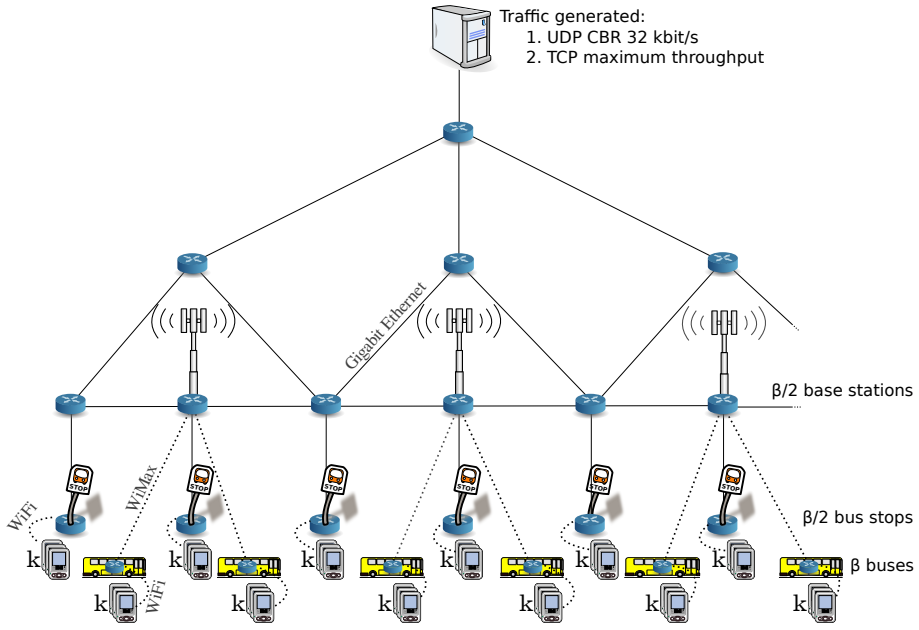


Figure 3.22: The "road scenario" network topology

Rbridges, interconnected with Gigabit Ethernet or fiber optic links. In our scenario, the spacing between bus stops is 1 km, as is the spacing between the row of base stations and the row of bus stops. We assume that the 802.16 base stations have a range (cell radius) of approximately 1.5 km, so that there is complete 802.16 coverage along the road. The capacity of a mobile 802.16 link varies considerably with the topographic conditions and distance to the base station, however some performance measurements put the 802.16 outdoors capacity at around 1.5–3 Mbit/s for the uplink direction, and 5–11 Mbit/s for downlink [63]. For the purpose of this study we will assume a (conservative) 802.16 capacity of 2 Mbit/s.

Along the road, a number (also β) of buses travel between bus stops. We recall that each bus contains one Rbridge, so they are the "moving network" part of the mesh network. There is a number (k) of terminals traveling inside each bus, and the same number of terminals waiting at each bus stop. The mobility model of buses is as follows. Each bus is initially stopped at a different bus stop, covering all bus stops. The buses travel from one bus stop to the next, stopping there for a period of time between 10 and 20 seconds, random and uniformly distributed. While stopping, all the terminals traveling inside the bus leave the bus and stay at the bus

stop, while at the same time the terminals in the bus stop enter the bus. The buses travel between bus stops in a realistic way, by beginning with a constant acceleration period of 10 seconds, followed by some distance traveled at a constant speed of 22.2 m/s, and finally a constant deceleration period of 5 seconds, right before stopping at the next bus stop.

There is a simple server node, near the Internet gateway, that is used for traffic generation purposes. In order to measure the routing protocol convergence as effectively as possible, we should consider only packets traveling downstream, from the server to each of the terminals. In the upstream direction, even if a terminal moves to another Rbridge, the route from the new Rbridge to the server is always the same because the server does not move. In the downstream direction, however, the Rbridge near the server has to know the new location of the terminal in order to ingress the packet into an MPLS tunnel ending in that Rbridge, and here is the true problem; here is where we need to focus our measurements.

To study the scalability of our routing protocol, we only need to increase the β parameter. The number of buses, β, is increased accordingly (3.1). By keeping k constant, the total number of terminals in buses and bus stops, T, also increases proportionally (3.2). The number of Base Stations, B, is also proportional to the number of Bus Stops (3.3). Likewise for the number of "tier-2" Rbridges (between base stations) (3.4) and "tier-3" Rbridges (3.5). Only the Internet Gateway and Server remain singleton as the network scales.

$$b = \beta \tag{3.1}$$
$$T = 2k\beta \tag{3.2}$$
$$B = \beta/2 \tag{3.3}$$
$$R_2 = \beta/2 \tag{3.4}$$
$$R_3 = \beta/4 \tag{3.5}$$

In our evaluation, we will focus on the 802.16 links of buses, which are the most resource constrained links (the remaining links being wired or WiFi) in the entire network. If routing overhead is acceptable for those links, then it will necessarily be acceptable for the entire network. As previously mentioned, in this thesis we only address the part of the routing protocol responsible for the mobility of terminals, namely MC messages.

Analytical model

The analytical model for WMRP base protocol in this scenario has already been mentioned. The rate of MC messages received on average by each node in each of its available links is given by $P = \frac{N-1}{\tau}$, with N the total number of Rbridges and τ the periodic MC refresh rate. The overhead bitrate (bit/s) is independent of the number of links. In this case, each MC message carries k MAC identifiers.

If μ represents the rate of MC message generation (1/MC interval), H the fixed header size of an MC message (20 bytes), and M the size of each entry inside the MC message (8 bytes), the global MC flooding overhead bitrate, in bit/s, of the protocol can be given by (not including the mobility triggered messages yet):

$$
\begin{aligned}
Overhead_{global} &= \mu\beta\,(H + (k+2)\,M) && (3.6)\\
&+ \mu b\,(H + (k+3)\,M) && (3.7)\\
&+ \mu R_2\,(H + 5M) && (3.8)\\
&+ \mu B\,(H + 5M) && (3.9)\\
&+ \mu R_3\,(H + 3M) && (3.10)\\
&\quad \text{(bit/s)}
\end{aligned}
$$

The overhead components can be explained by considering each type of Rbridge and the fact that even Rbridges themselves are seen as terminals by neighboring Rbridges. The first term (3.6) gives the overhead of bus stops, which sees k end user terminals, plus one Rbridge as terminal, plus one bus connected to it (on average). The second term (3.7) represents the overhead generated by buses, and it includes k terminals, plus one MC entry for another bus on average connected to the same base station, plus an MC entry for the bus stop WiFi link to which the bus is (at least part of the time) connected, and yet another entry for the base station to which it is connected. The third term (3.8) denotes the overhead generated by tier-2 transit Rbridges, which have 5 neighbors each. The fourth term (3.9) includes the overhead generated by base stations, which have 3 links each to neighboring fixed Rbridges, and on average are two more links to moving Rbridges inside buses. Finally, the term (3.10) expresses the overhead generated by tier-3 Rbridges, with 3 links each to report. The expression can be condensed as:

$$Overhead_{global} = \frac{\mu\beta}{4}\left(11H + (8k+33)\,M\right) \tag{3.11}$$

For the *explosive* mobility optimization, the overhead model is similar to the base protocol overhead with $\tau = 60$; we only have to add the additional overhead caused by the BU and the TTL-limited MC flooding. It can be shown that, for *explosive* with TTL limit of 2, the additional WMRP messages that pass through the bus 802.16 links are $6k$ MCs, $2k$ BUs, and $2k$ BAs[7]. In this case, the additional mobility-related MC messages carry a single MAC identifier each.

We also need to take into account the time it takes for a bus to travel between bus stops, which is given by (3.16), where t_s denotes time a bus is stopped on a bus stop (20 seconds), t_a denotes the acceleration period (10 seconds), t_d the deceleration period (5 seconds), d_t the total distance traveled between bus stops (1000 meters), d_a the distance traveled while accelerating, d_d the distance traveled while decelerating, and s the top speed of the bus (22.22 m/s, 80 km/h).

$$a = s/t_a \tag{3.12}$$
$$b = s/t_d \tag{3.13}$$
$$d_a = \frac{1}{2}at_a^2 \tag{3.14}$$
$$d_d = st_d - \frac{1}{2}bt_d^2 \tag{3.15}$$
$$t_{travel} = t_s + t_a + \frac{d_t - d_a - d_d}{s} + t_d \tag{3.16}$$
$$= t_s + \frac{t_a + t_d}{2} + \frac{d_t}{s} \tag{3.17}$$

The overhead due to handover for the *explosive* case can then be given by:

$$Overhead_{explosive} = Overhead_{global} + \frac{6kH_{MC} + 2kH_{BU} + 2kH_{BA}}{t_s + \frac{t_a+t_d}{2} + \frac{d_t}{s}} \tag{3.18}$$

[7]BA stands for Binding Acknowledgment and is used to confirm the correct reception of a previous BU message.

In (3.18), H_{MC} represents the size of a MC message with one entry (28 bytes), H_{BU} is the size of a BU (Bindging Update) message (40 bytes), and H_{BA} the size of a BA (Bindging Acknowledgment) message (24 bytes). For instance, for $k = 10$, replacing the values we obtain:

$$Overhead_{explosive} = Overhead_{global} + 326.6 \text{ bit/s} \quad (3.19)$$

Clearly, the 326.6 bit/s component is constant and $Overhead_{global}$ will easily surpass it for medium-to-large networks by several orders of magnitude, thus we may make the approximation that $Overhead_{explosive} \approx Overhead_{global}$. If we were to develop an analytical expression for $Overhead_{bindupdate}$, we would draw a similar conclusion.

Simulation Setup

For simulations, we used the ns 3.2 simulator, with additional models and back-ported bug fixes. At the link layer, the simulations were configured as follows. For 802.11, a constant data rate was set to 11 Mbit/s, rather than the adaptive one, and a simplified 802.11 MAC layer was used. The new *statistical MAC layer* simulates the 802.11 MAC service by means of a random number generator that provides transmission delays according to a log-normal probability distribution, which closely resembles the 802.11 MAC delays, congruent with what had already been concluded analytically in [64]. This technique makes our simulations run about five times faster, at the cost of a certain loss of precision. But since we are not trying to discover what is the exact capacity of the simulated network, rather just observe how the routing protocol scales, simulation precision is less important than being able to simulate large networks. The simplified 802.11 MAC requires the definition of a hard cutoff range for the signal, which has been defined as 30 meters for the AP inside buses, and 100 meters for the AP in the bus stop. These are the approximate indoor and outdoor ranges, respectively, of typical WiFi 802.11b/g equipment. We also used the statistical WiFi MAC to simulate the 802.16 links, configured for a range of 1500 m and a bitrate of 2 Mbit/s. This method has less precision than the builtin ns-3 802.16 module, but allowed us to simulate much larger networks.

A large set of simulations were run, varying the number of bus stops between 16 and 256, and setting the k parameter to 2. Although $k = 2$ may seem limiting, it allows us to simulate larger networks. The analytical model considers other k values (see Fig. 3.27). Thus, the number of mobile terminals varied between 64 and 1024. Each simulation simulates 600 seconds,

and was repeated 3 times for confidence interval purposes[8]. Additionally, at each network size the following configurations were simulated in turn:

static60 Consists in grounding all buses and mobile terminals, so that there is no mobility. The MC refresh period is set to 60 s. This simulation is used to establish the top-line of the results (the maximum network performance is expected to be attained in a static network);

base5 The base WMRP protocol, configured with OLSR settings, i.e. MC refresh period of 5 s. Mobility of both buses and terminals is present, but no terminal mobility optimizations enabled. As noted in Sec. 3.2.4, for all intents and purposes we may consider this the same as OLSR;

base60 Base WMRP with MC refresh period of 60 s;

bindupdate The "binding updates" optimization, with mobility enabled and a 60 s MC refresh period;

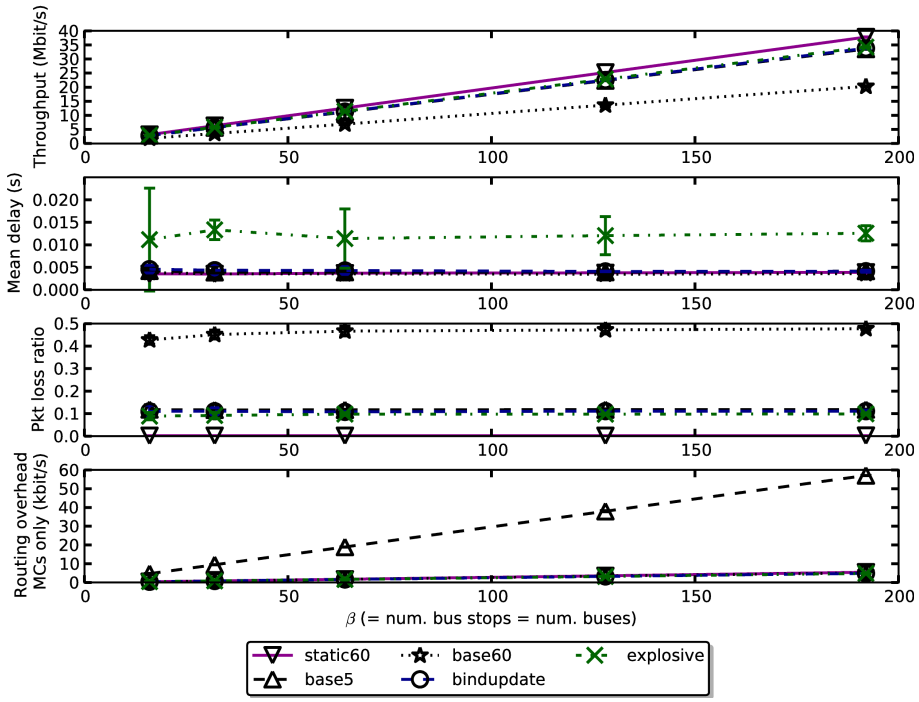
explosive The "explosive updates" optimization, with mobility enabled, 60 s MC refresh period, and TTL=2 for the mobility MC updates.

We did two sets of experiments with different traffic patterns. In one set we simulate the server transmitting a CBR UDP flow to each terminal. Each flow consists of 4 packet/s, with packet size 1000 bytes, totaling 32 kbit/s. In another set of experiments there is one TCP connection between the server and each terminal, and we attempt to transmit as much data as possible through each of these connections.

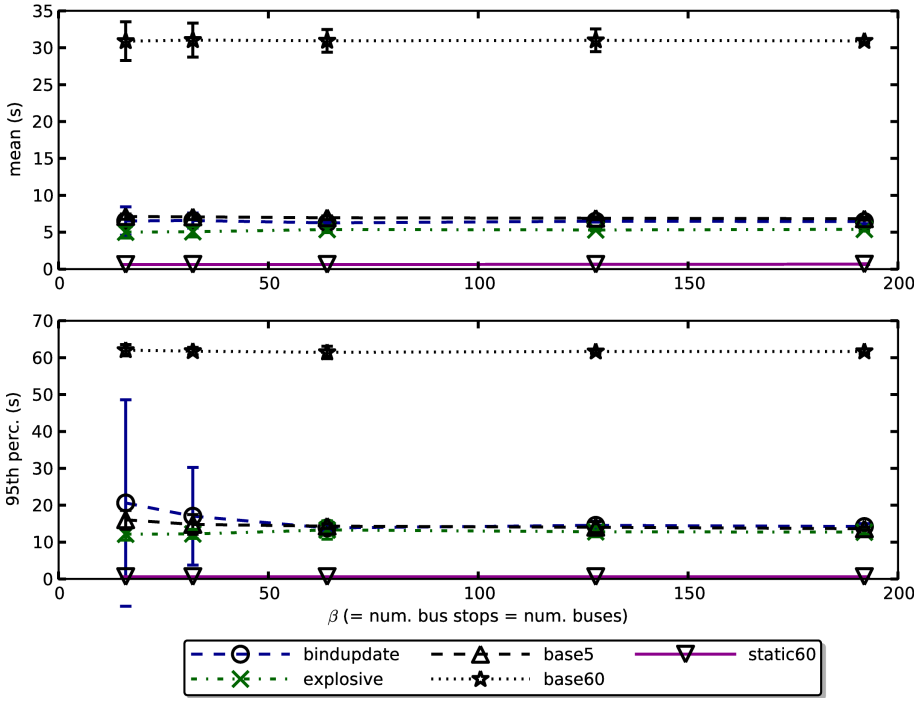
Simulation Results

The network performance results are shown in Fig. 3.23 (UDP) and Fig. 3.24 (TCP). All the curves in all plots have represented confidence intervals, although they are too small to see in some cases. The top subplots show the received bitrate, end-to-end delays, packet losses, and routing overhead. All values are averaged over all flows, except the bitrate which is the sum of all flows, and the routing overhead which is independent of the flows and is the total bitrate that passes on the WiMax link of each bus. The bottom subplots show what we call "flow interruptions". We consider there is a flow interruption whenever the inter-arrival time of received packets in

[8]From the obtained results, due to the large number of packets processed in each simulation, we have found that repeating 3 times each simulation was enough to obtain a very small 95% confidence interval for large networks, as can be seen in the results figures.

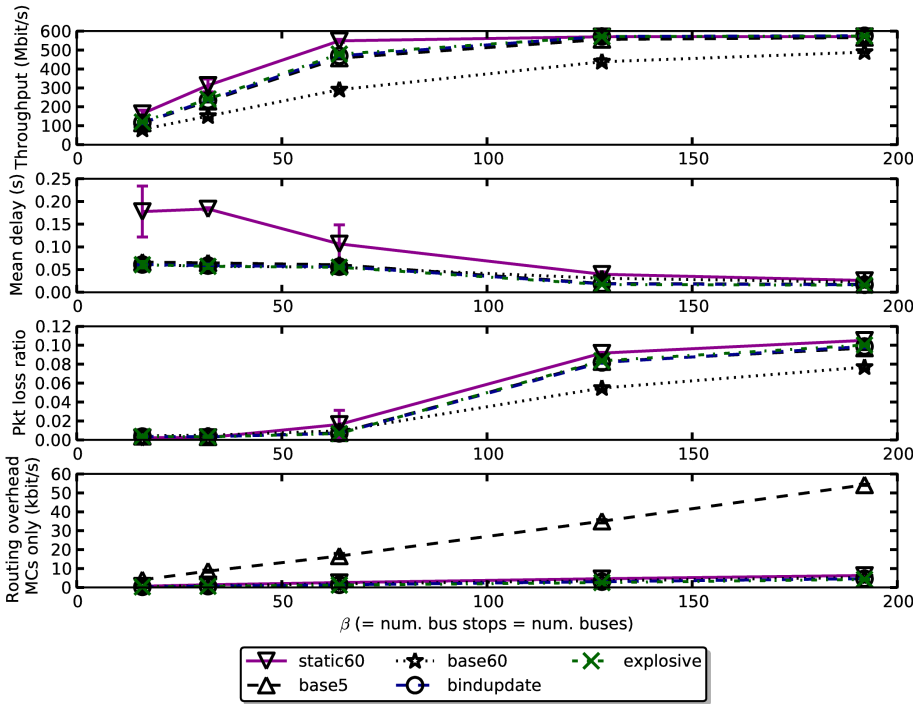


(a) Throughputs, delays, losses, and routing overhead

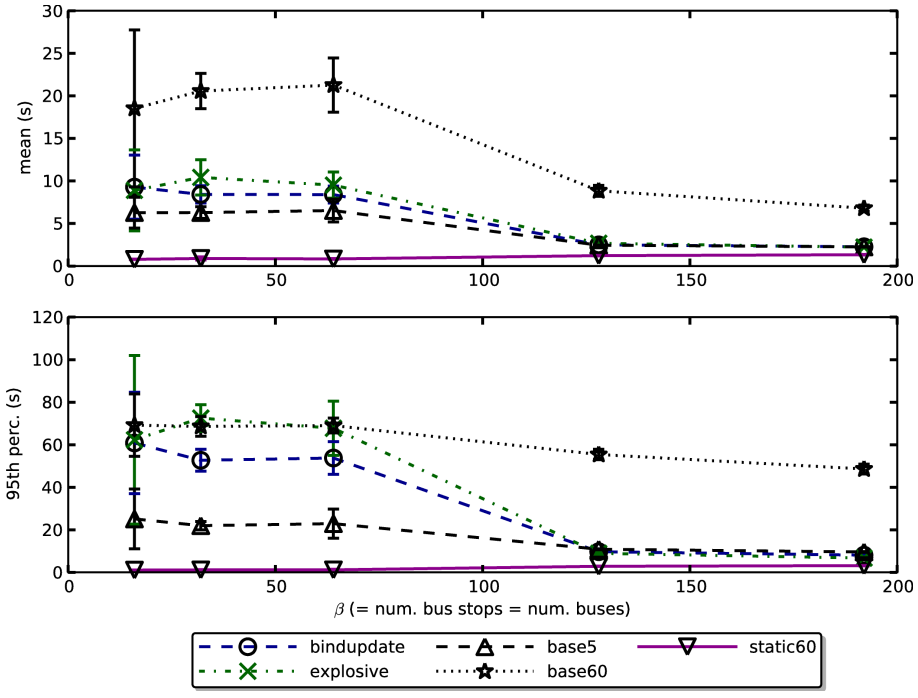


(b) Flow interruptions

Figure 3.23: Road scenario: UDP results



(a) Throughputs, delays, losses, and routing overhead



(b) Flow interrutpions

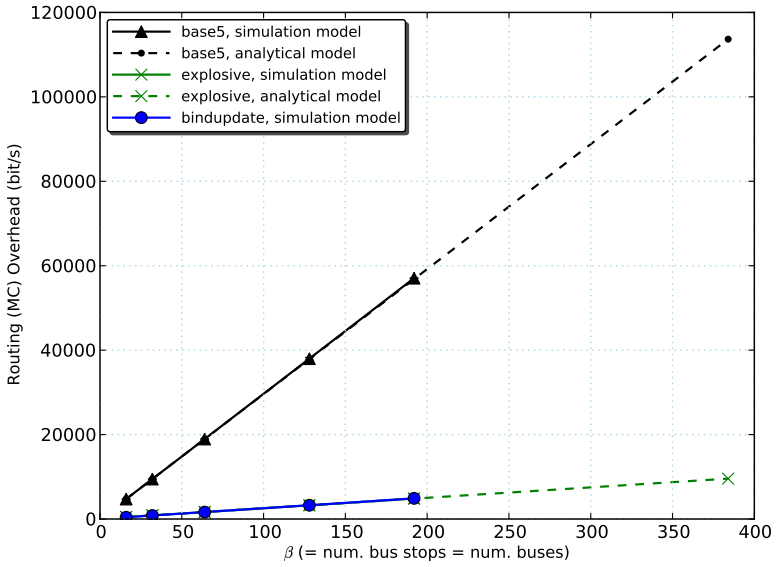
Figure 3.24: Road scenario: TCP results

the terminal exceeds 0.5 seconds. Whenever such an interruption occurs, the duration of the interruption is recorded in a histogram. In the bottom subplots we show the mean, 95th percentile, and sum of all flow interruptions over all flows. These flow interruptions effectively measure the time taken by handovers from the point of view of the applications, and is directly related to the time it takes for the routing protocol to discover new accurate routes for moving terminals.

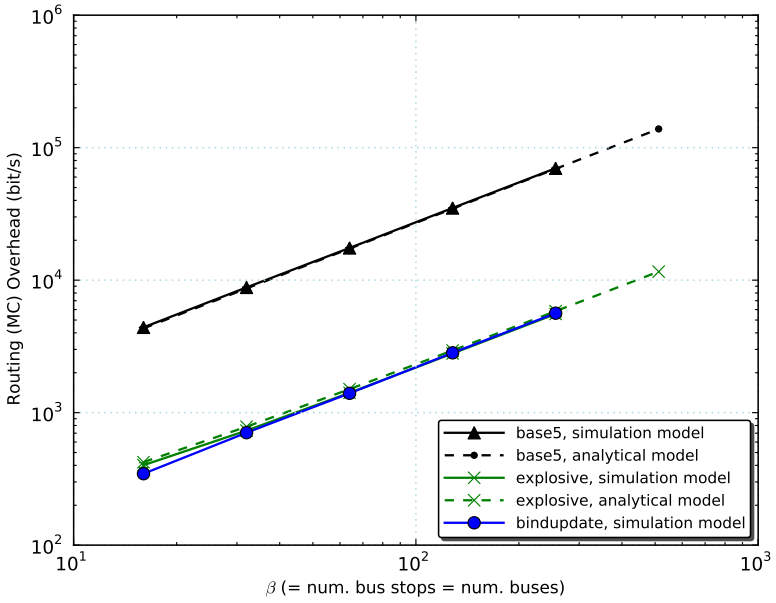
From the presented results we may draw a number of conclusions. First, our mobility optimizations, *bindupdate* and *explosive*, achieve significantly better results, in terms of bitrate and packet loss ratio, than *base60* while generating the same routing overhead. Another comparison we can make is that the mobility optimizations achieve identical user plane results as *base5* but using only a fraction of the control plane overhead. These conclusions hold also for the "flow interruptions" metric, and for either UDP or TCP traffic.

The explanation for these results is simple. The base60 solution has low routing overhead because it causes Rbridges to send only one MC every 60 seconds. But this very slow refresh period also causes routes to be frequently out of date, hence the greater packet losses and lower bitrate. On the other hand, *base5* has frequently updated routes and good user plane performance, but all the routing updates are global and so the routing overhead will be high. Our solutions have slow global updates, hence the low routing overhead, but frequent localized routing updates, which do not have global impact on the total routing overhead but provide updated routes very quickly, hence the low routing overhead and good user plane performance.

If we would consider only the user plane results seen so far, it would seem like a simple base WMRP configuration with 5-second refresh interval (like the OLSR default TC or HNA refresh interval) is about as good a solution as the terminal mobility optimizations described in Sec. 3.5.1 and Sec. 3.5.2. However, this is not a complete picture. Fig. 3.25 shows the routing overhead obtained via simulation, together with the predicted values from the analytical models. We can see in these results that *base5* has at least an order of magnitude greater routing overhead. It can also be observed that all the mobility solutions eventually tend to a linear growth, for large networks. The non-linearity in *explosive* for small networks is due to the fixed overhead introduced due to mobility, but that fixed overhead eventually becomes insignificant with increasing network size. Finally, these results also show that the analytical and simulation models are in agreement.



(a) Linear



(b) Logarithmic

Figure 3.25: Road scenario: routing overhead (MCs only) simulation and analytical results: linear scale (a) and logarithmic scale (b).

Network scalability limit

With the results so far, the main quest for discovering the scalability limits of WMRP is yet to be completed. Via simulation we have confirmed the intuitive assumption that the routing overhead, on a per-link basis, scales linearly with the network size. To find out whether a routing protocol scales to a certain network size, ideally we would like to simulate a very large network until the routing overhead exceeds a certain limit. Unfortunately, simulating such a complex system takes a toll on computing resources and we have found it difficult to simulate networks larger than 256 bus stops / 1024 mobile terminals. Instead, we will utilize analytical models to predict results for any network size without simulations. In this study, we consider 5% of the 802.16 link capacity (100 kbit/s) as an acceptable limit for the maximum bandwidth consumed by the MC messages. No packet rate limit is considered because the WMRP agent already takes care of aggregating multiple messages into a small number of large packets.

Fig. 3.26 presents the results of the scalability analysis for two values of k: 2, and 10. From these curves, we can obtain the following limits. For $k = 2$, the base protocol crosses the 5% capacity limit at $\beta = 320$ (1280 terminals[9]), but with the optimizations in effect the limit is crossed at $\beta = 5006$ (20024 terminals). For $k = 10$, the base protocol crosses the 5% capacity limit at $\beta = 166$ (3320 terminals), but with the optimizations in effect the limit is crossed at $\beta = 2590$ (51800 terminals). In both cases, the optimization increases the number of bus stops and mobile terminals by a factor of approximately 16. In the case of the *explosive* mobility optimization, these limits are independent of the user traffic pattern.

According to equation (3.11), the maximum β limit is a direct function of k and μ, and μ is the inverse of the periodic MC generation interval. By replacing the overhead with the 5% limit, 100 kbit/s, and solving the equation for β as function of k and μ, we may obtain a scalar field. In Fig. 3.27 the evolution of the maximum β is shown for a range of k and "MC interval" values. As expected, a greater MC interval will cause less routing overhead to be generated, and so allows β to attain larger values, i.e., larger network size. However, a too large interval between global updates would bring other problems, such as the increased transient state that Rbridges have to keep due to terminal mobility. On the other hand, a larger k value (terminals per bus or bus stop) will cause each MC message to carry more MAC identifiers and so generate more routing overhead, leading to a reduced network size. The k parameter cannot be controlled, but it is not expected

[9]We recall that the number of terminals, T, is given by $T = 2k\beta$.

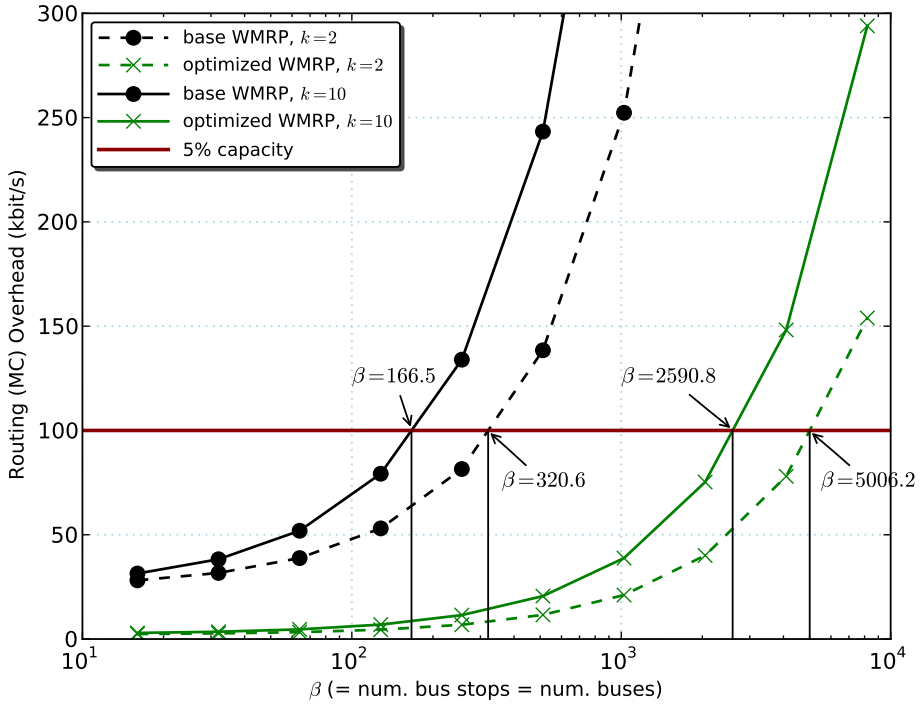


Figure 3.26: Road scenario: routing protocol scalability prediction

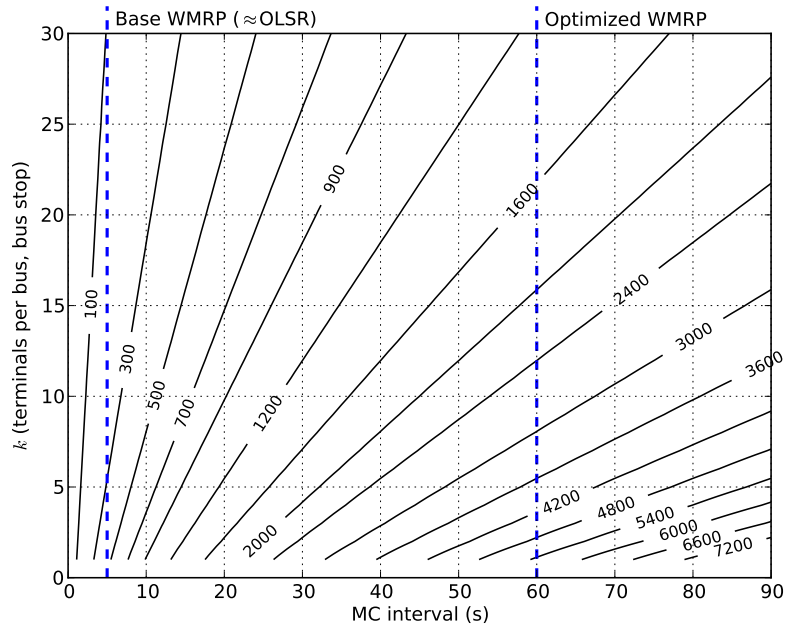


Figure 3.27: Maximum β for a range of k and "MC interval" values

to be large; a large number of active terminals are expected, but they should naturally spread among many different buses or bus stops (or else even the WiFi network will be overloaded).

3.6.2 City grid scenario

Scenario

We consider the scenario of a city organized as a grid of city blocks, exemplified in Fig. 3.28, for the case of a very small network. Each block has the typical "Manhattan" city block size of 80 by 274 m, which is a prototypical size followed in many cities around the world. Between city blocks we consider that there exist roads for vehicles, and at each road intersection we assume the presence of a traffic stop sign. There are a number of bus stops spread throughout the city grid, at the ratio of one bus stop every 2 blocks vertically (we consider the larger side of each city block the vertical side), and one every 4 blocks horizontally.

At the beginning of the simulation, a number of buses exist at each of

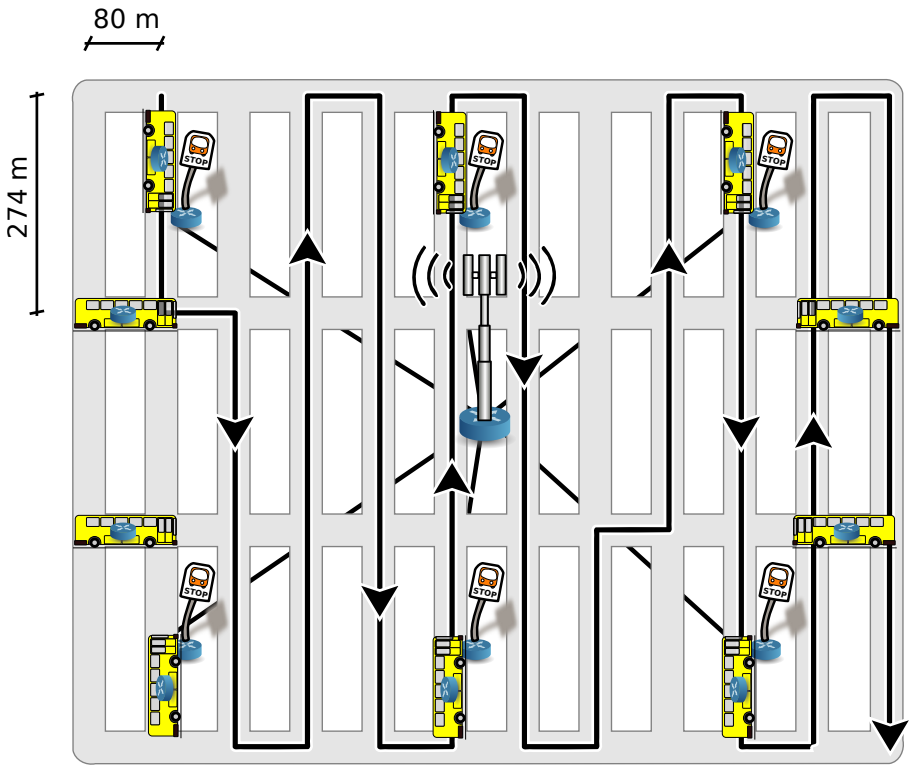


Figure 3.28: Example "city grid" topology, showing some buses at their initial positions and the path of one of those buses.

the four edges of the city grid. At each of the north and south edges we have 1/4 as many buses as the horizontal number of city blocks, while at the east and west edges we have half as many buses as the vertical number of blocks. The buses cross the city center traveling along existing roads from one edge to the other, stopping at each intersection, due to the traffic stop signs, in conformance with a Stop Sign Mobility Model [65]. They will also stop for 20 seconds at each bus stop along the path, allowing the passengers to enter and leave the bus. A sample path taken by a bus is shown by the solid line with arrows in Fig. 3.28; as seen there, the bus route is vaguely a zig-zag line, but constrained to the city roads, ensuring that the whole city area is covered.

The city area has full WiMax coverage, provided by grid of equally spaced base stations. The WiMax bandwidth and range vary greatly with the topographical conditions where it is deployed, as well as frequency band. In this suburban scenario we consider a bandwidth of 2 Mbit/s and range of 700 m. To provide full coverage, it can be shown that the base stations have to be placed in a grid topology, spaced by $\frac{2\times 700}{\sqrt{2}} = 990$ m. All the base stations are connected in a grid topology by Ethernet links, and each bus stop is linked to the nearest base station.

There is an Internet server connected to the WiMax base station closest to the center of the grid by 1 Gbit/s Ethernet, while we have 5 mobile terminals attached to each bus and each bus stop. Every time a bus arrives at a bus stop, the 5 terminals inside the bus leave it for the bus stop, while the 5 terminals previously at the bus stop enter the bus. Although 5 terminals per bus and bus stop may seem too few, it allows us to simulate slightly larger networks. However, the analytical model shown further down expands the scope of the main results to a range of terminals per bus and bus stop between 1 and 30 (see Fig. 3.32).

Analytical model

As in the case of the road scenario, the routing overhead has essentially two components: 1) a variable component that grows with the network size, and 2) a fixed component that is independent of the network size (in case of bindupdate and explosive solutions). In similar way to what was shown in Sec. 3.6.1 for the "road scenario" , as the network grows the variable component also grows proportionally, and the total routing overhead becomes dominated by that parcel. To simplify the analysis, we disregard the fixed component and consider only the variable component. This simplification will be validated by comparing against simulation results. In these equa-

tions, the $\lceil x \rceil$ notation denotes a *ceiling* operation, i.e. round to the nearest integer larger or equal to x.

As before, we want to find out the amount of MC overhead in the weakest link, i.e. the received bitrate in buses' WiMax links. This overhead is essentially proportional to the total number of Rbridges, and the number of registered terminals in each of those Rbridges. The number of each type of Rbridge is a function of the city area, measured in blocks. To simplify, we consider an area of $h \times h$ blocks.

The bus stops are laid out in a grid, $h/4$ by $h/2$, and so the number of bus stops can be given by:

$$S = \left\lceil \frac{h}{4} \right\rceil \times \left\lceil \frac{h}{2} \right\rceil \tag{3.20}$$

Along the north and south edges of the grid, each edge with $h+1$ intersections, there are at least $(h+1)/4$ buses. We add the east/west borders, with at least $(h+1)/2$ buses each. Therefore, the total number of buses can be given by:

$$\beta = 2 \left\lceil \frac{h+1}{4} \right\rceil + 2 \left\lceil \frac{h+1}{2} \right\rceil \tag{3.21}$$

To compute the number of WiMax base stations we have to consider that they are displayed in a grid and they need to cover the entire city area. The spacing of base stations has to be 990 meters, and the city grid has total dimensions given by the Manhattan city block size multiplied by the number of blocks: $h \times 80$ and $h \times 274$. We add one row and one column of base stations to make sure the edges are covered. Hence, the number of base stations is given by:

$$B = \left\lceil 1 + h\frac{80}{990} \right\rceil \times \left\lceil 1 + h\frac{274}{990} \right\rceil \tag{3.22}$$

The overall number of Rbridges is given by the sum of the expressions (3.20), (3.21), and (3.22). It is easy to see that this sum can be represented as the polynomial expression $C_1 + C_2 h + C_3 h^2$, with C_1, C_2, and C_3 constants. Denoting by A the city area, since $A = h^2$ we may represent the expression as $C_1 + C_2 A^{\frac{1}{2}} + C_3 A$. Considering the asymptotic behavior, we can see that $O\left(C_1 + C_2 A^{\frac{1}{2}} + C_3 A\right)$ simplifies to $O(A)$ under "big O" notation rules. In other words, the total number of Rbridges tends to grow approximately linearly, for large grid sizes.

As an example, consider the case of $h = 20$. The grid size would be $20 \times 20 = 400$ city blocks, with an euclidean area equal to $(20 \times 80) \times (20 \times 274) = 8.77$ km^2. We would then have 34 buses, 50 bus stops, and 21 base stations. If we would consider to have 5 terminals per bus or bus stop, there would be a total of 420 end-user mobile terminals.

To compute the average number of bus stops connected to each base station, S_B, we multiply the total number of buses by the ratio of base station coverage over total city area, and we obtain eq. (3.23). Similar reasoning applies to eq. (3.24), which computes the average number of buses in range of each base station, β_B.

$$S_B = \left\lceil S \frac{990^2}{h \times 80 \times h \times 274} \right\rceil \tag{3.23}$$

$$\beta_B = \left\lceil \beta \frac{990^2}{h \times 80 \times h \times 274} \right\rceil \tag{3.24}$$

Finally, we can compute the total overhead produced by Rbridges periodically generating MC messages. Denoting by H the header size of the MC message (20 bytes), and by M the size of each entry inside the MC message (8 bytes), T_S the number of terminals per bus stop (5, in the simulations), and T_β the number of terminals per bus (5, in the simulations):

$$\begin{aligned} \text{Overhead} &= \mu S \left(H + (T_S + 1)\, M\right) && (3.25) \\ &+ \mu \beta \left(H + (T_\beta + 1 + T_B - 1)\, M\right) && (3.26) \\ &+ \mu B \left(H + (S_B + \beta_B)\, M\right) && (3.27) \end{aligned}$$

In the above expression, μ represents the rate of MC message generated, i.e. 1/MC interval. The subexpression (3.25) represents the overhead resulting from MC messages generated by bus stops, (3.26) the MC overhead generated by buses, and (3.27) the MC overhead generated by base stations.

Simulation Setup

To evaluate the city grid scenario, described in Sec. 3.6.2, we ran a series of simulations where the grid size is gradually increased. Two sets of simulations were performed. In one set, one 32 kbit/s UDP flow (packet size 1000 bytes) was generated by the Internet server, directed to each mobile terminal. In another set of simulations, there is one TCP connection between the Internet server and each mobile terminal, and the server tries to

transmit as much data as possible over the TCP connection to the clients. The mobile terminals handover between bus and bus stop, and vice-versa, as previously described, and general flow statistics are captured, using the ns-3 Flow Monitor module (see Sec. 4.2), as well as routing protocol overhead.

Simulation Results

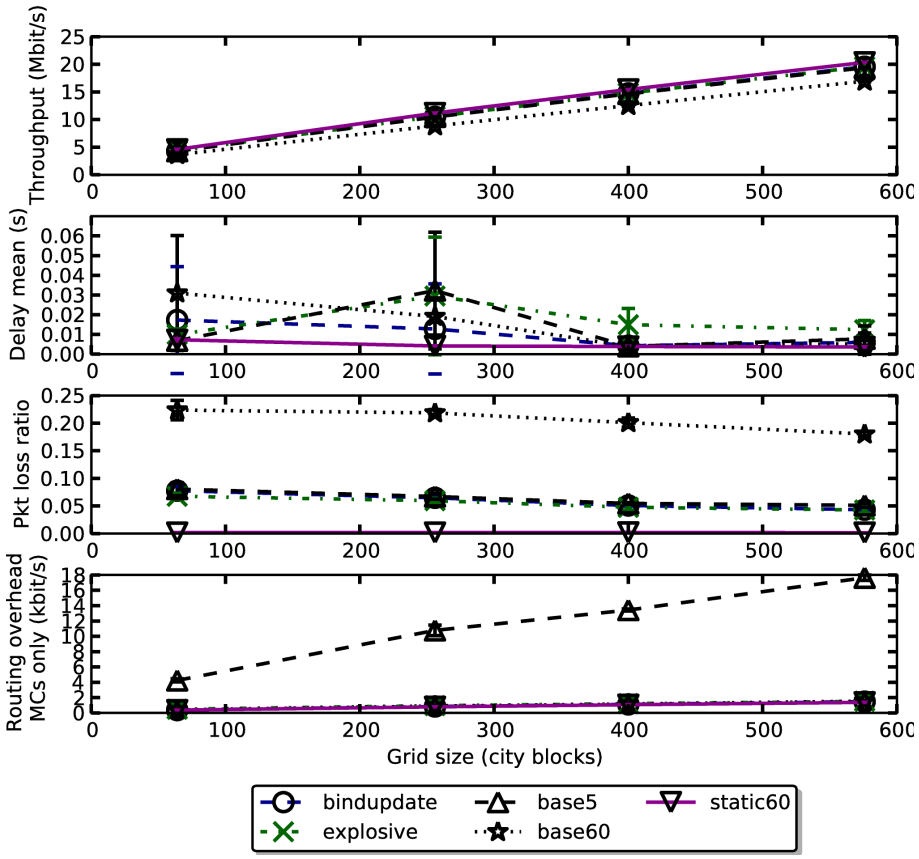
Fig. 3.29 and Fig. 3.30 show some of the simulation results for the city grid scenario with UDP and TCP traffic, respectively. Here we can see that the *base5/bindupdate/explosive* solutions have nearly indistinguishable performance metrics, whilst the *base60* solution has marginally less throughput and significantly higher packet loss ratio. While the gain of the WMRP optimizations is small for the bitrate/loss/delay metrics, in the flow interruptions metrics the gain is very clear. The *base60* solution has about 22 seconds of mean interruption time, for the UDP case, while the others (bindupdate/explosive) have 5 to 6 seconds mean interruption. In the TCP case, *base60* has about 6 seconds mean interruption time, while the optimized solutions have about 2.5 seconds mean interruption. The lower value of mean interruption time in TCP, compared to UDP, may be explained by the apparent flow interruptions that are in fact not cause by handover but by TCP congestion control taking some time to adjust to mobility.

The evolution of the routing protocol overhead with increasing city grid area is shown in Fig. 3.31. Both analytical and simulation results are represented in the same plot, allowing us to confirm that the analytical model is accurate for large networks. These routing overhead results are obtained for the simulations with UDP traffic, but with TCP traffic the results are identical.

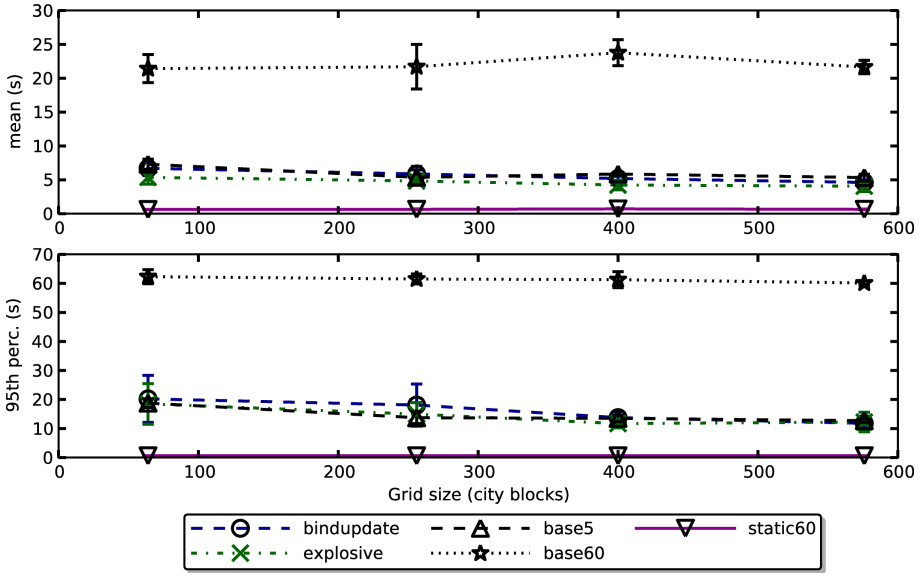
As expected, the *base60* solution generates a small fraction of the overhead of *base5*. Additionally, *explosive* and *bindupdate* have the same control plane overhead as *base60*, but have as good (if not better) user plane behavior as *base5*. Thus we conclude that the optimized solutions have the best of both worlds: good user plane performance, but with a low control plane cost.

Network scalability limit

From Fig. 3.31 we can find out, for each WMRP configuration, what is the grid area size for which the 5% WiMax capacity limit is crossed. Thus we can see that the non-optimized *base5* solution (similar to OLSR) scales to 4983 city blocks (71×71), which is equivalent to an area of 124 km^2. With

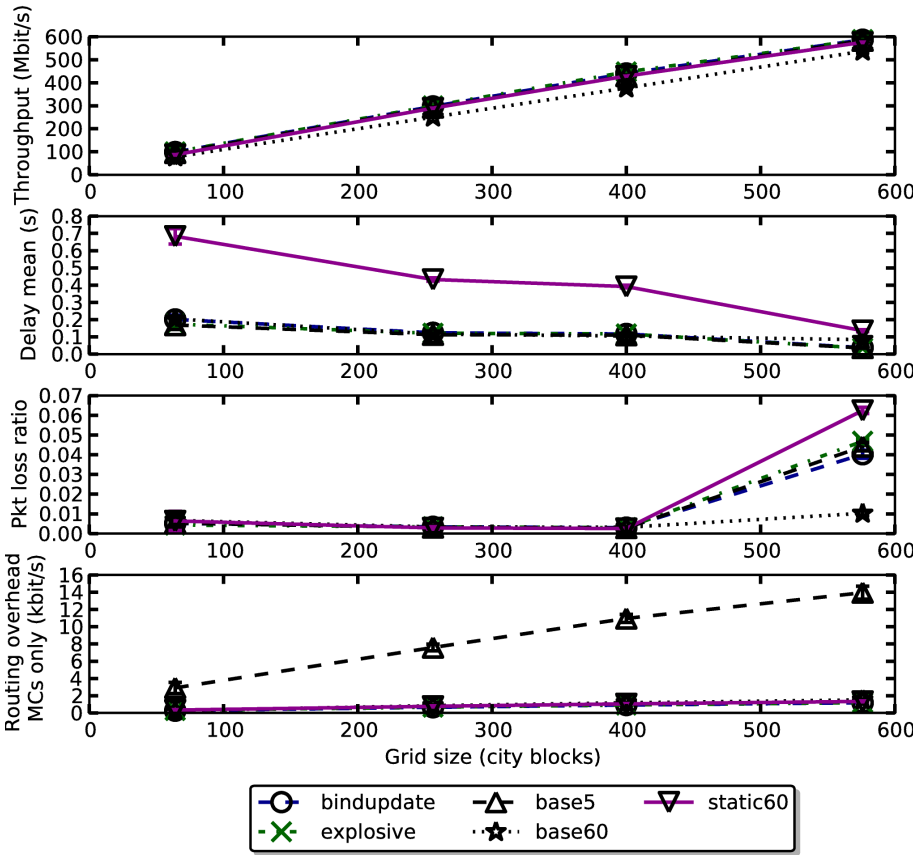


(a) Throughputs, delays, losses, routing overhead

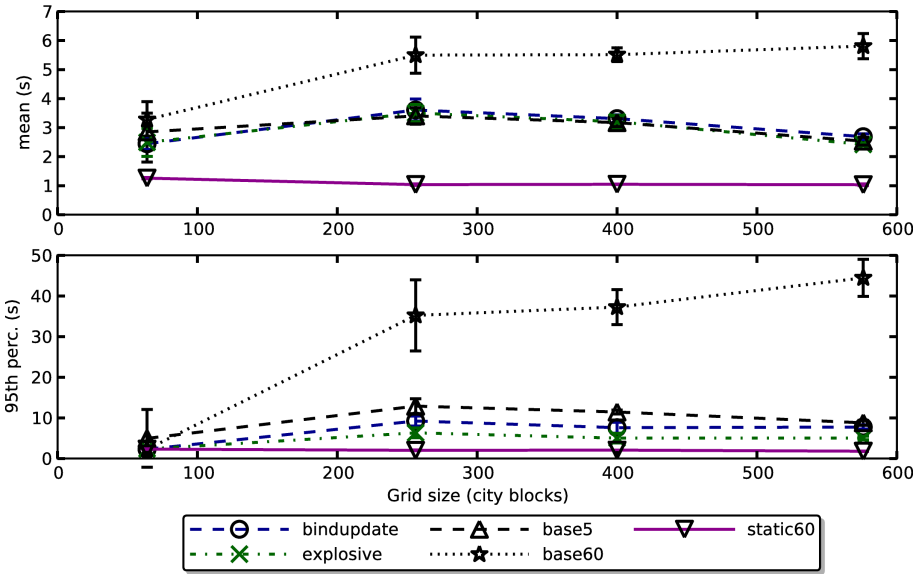


(b) Flow interruptions

Figure 3.29: City grid scenario: UDP results



(a) Throughputs, delays, losses, routing overhead



(b) Flow interrutpions

Figure 3.30: City grid scenario: TCP results

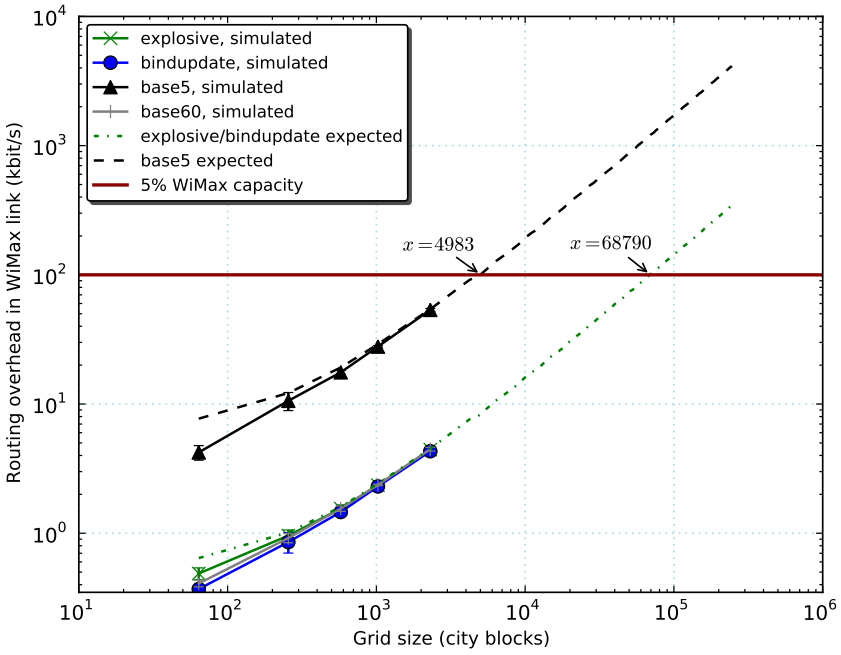


Figure 3.31: Grid scenario: routing protocol scalability prediction

the network size at which the threshold is reached, the scenario predicts 630 bus stops, 108 buses, 147 WiMax base stations, and 3690 mobile terminals.

With either of the optimized *explosive/bindupdates* solutions, the 5% capacity threshold is crossed at much larger network size of 68790 city blocks (262×262), or 439 km^2. At the maximum network size, we would have 8646 bus stops, 396 buses, 1702 base stations, and 45210 mobile terminals. Compared to the non-optimized solution, the maximum city area increases by a factor of 3.54, and the number of mobile terminals increases by a factor of 12.25.

The obtained limit depends essentially on two parameters: μ and k, considering the simplification that $T_S = T_\beta = k$. Considering that μ is the inverse of the "MC interval", it is easy to determine what is the maximum city grid size that can be attained without surpassing our imposed overhead limit of 100 kbit/s, for a range of MC interval and k values. The obtained scalar field is represented in Fig. 3.32. As in the road scenario, we can see that the maximum network size increases when the MC interval increases and when k decreases.

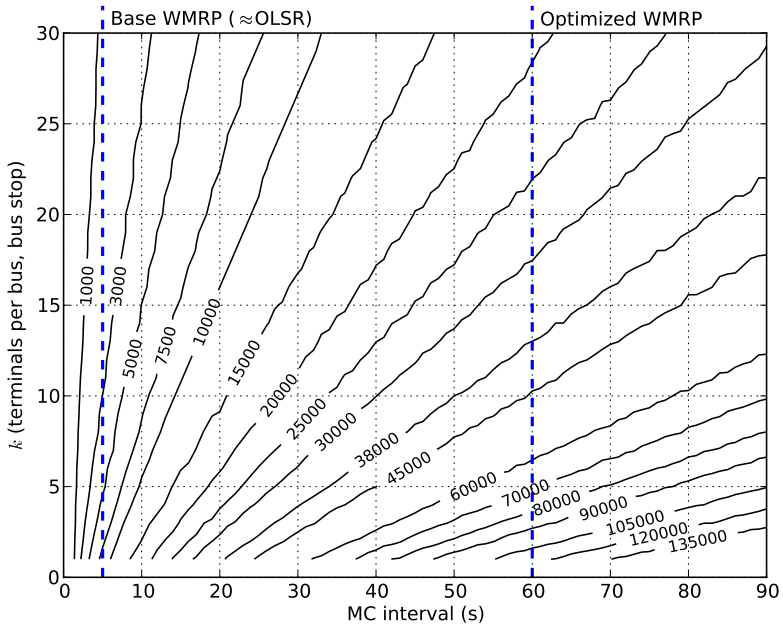


Figure 3.32: Grid scenario: maximum size (in city blocks) for a range of k and MC interval values

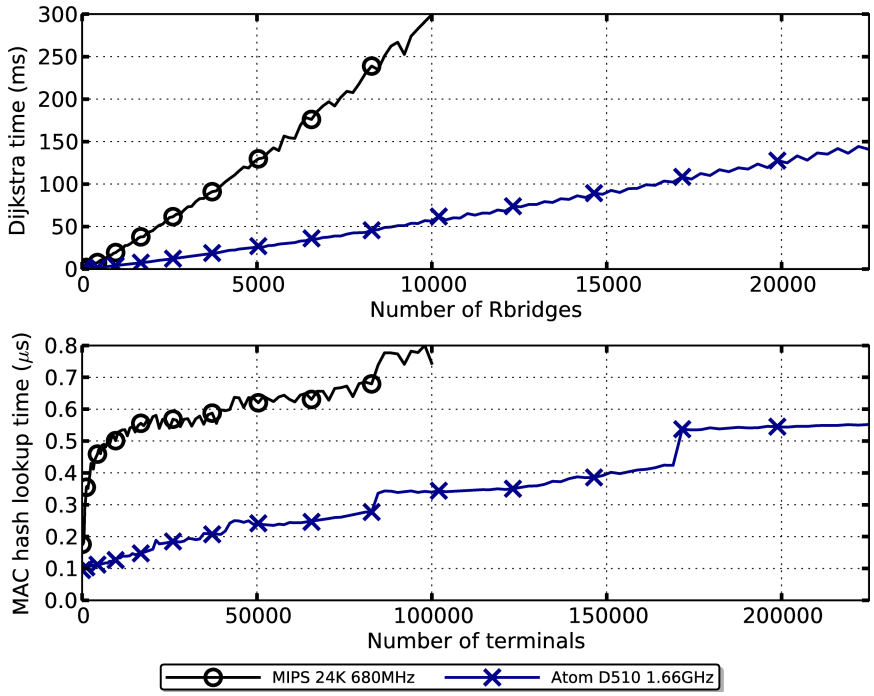


Figure 3.33: Benchmark of Dijkstra's algorithm (top), and MAC-48 hash table lookup (bottom)

3.6.3 Computational scalability

The WiMetroNet control plane is designed to scale to many thousands of Rbridges and tens of thousands of end user terminals. However, it is not clear how well will a hypothetical implementation handle all the necessary operations in that scale. As possible implementation targets, we are considering two possibilities. The first target is simple DD-WRT wireless router platform, based on *RouterStation Pro Board* containing an Atheros AR7161 MIPS 24K CPU running at 680MHz with 128MB of DDR memory. The second target being considered is based around a more powerful Intel processor, specifically the Intel Atom D510 running at 1.66GHz.

Regarding the Rbridge data plane, ingress of end user frames is clearly the most difficult operation, specifically the look-up of a destination MAC address in the *Remote Terminal Associations* table. To evaluate MAC address look-up performance, we executed benchmarks on a simple program that does hash table look-ups with random MAC-48 addresses as keys. The results in Fig. 3.33 (bottom) show that hash table look-ups are below $1\mu s$

even for the MIPS CPU and tables with 100,000 entries.

With respect to the Rbridge control plane, the most expensive operation is expected to be the shortest-path computation. In the WMRP routing agent, whenever a TC message arrives, and it represents a change in the topology (rather than e.g. just refresh an existing topology tuple), a flag is set indicating that new shortest paths need to be computed. Once every 250 ms the WMRP agent checks this flag, and if it is set, Dijkstra's shortest path algorithm is run. Thus, in order for the WMRP agent to keep up with topology changes it needs to be able run Dijkstra faster than 250 ms for topologies of 13766 and 10744 Rbridges (the number of Rbridges needed for the maximum network size derived in the road and grid scenarios, respectively). To test this hypothesis, we benchmarked Dijkstra's algorithm on a grid topology while increasing the number of nodes between 100 and 22500 nodes. As shown in Fig. 3.33 (top), the MIPS CPU reaches the 250 ms limit at approximately 8000 nodes, while the Atom still is very far from reaching it at 22500 nodes. Thus, in an implementation based on the MIPS CPU, shortest path computation becomes a bottleneck that limits the scalability of WiMetroNet, but a router based on the much faster Atom CPU solves this problem. An alternative would be to make use of dynamic shortest path algorithms. In [66] several dynamic shortest path algorithms are evaluated and concluded to be 10 to 10,000 times faster than repeated application of a static algorithm.

3.7 Related work

This section discusses some existing solutions that, on the surface, seem to address a similar problem as WiMetroNet, and tries to highlight some key differences found.

The IETF Transparent Interconnection of Lots of Links (TRILL) [24] is working towards a standard solution for shortest-path frame routing in a multiple-hop 802.1-compliant network with arbitrary topology. For that purpose, TRILL proposes the concept of Routing Bridge (Rbridge), a node running a link-state routing protocol at L2. Other goals of the solution are: minimal configuration, routing loop mitigation (through the use of a TTL field) and legacy node support. However, TRILL lacks efficient and scalable mobility support, which is required in the WiMetroNet scenario. Additionally, because TRILL targets perfect compatibility with the service offered by legacy bridges, it does not limit broadcasts in general, and so does not scale to large network sizes. TRILL does optimize ARP and DHCP,

but there are other services that also use broadcast or multicast, such as NetBIOS, UPnP, and DNS-SD. These are not needed in a metropolitan area, but their use can take down a network, intentionally or not ("denial of service" attacks).

The LANMAR [67] routing protocol targets large scale adhoc networks. To accomplish its scalability goals, LANMAR nodes are organized into subnets. Each node has two logical identifiers, a subnet identifier, and a node identifier unique within its subnet. Within each subnet, one of the nodes is elected as the "landmark" node. When a packet from one subnet targets a node in another subnet, the packet follows the path to the landmark node instead of the end node, until it enters the target subnet, at which point the path follows the most direct route to the target end node. The drawback of the LANMAR approach is that it requires prior assignment of nodes into logical subnets considering how the nodes are naturally grouped and are likely to move. The example given for LANMAR is one of a military structure, with tanks and other units orbiting the tanks. This approach does not work well when no a priori structure of the moving nodes can be defined, or when the network topology is highly dynamic. LANMAR handling of "drifters and isolated nodes" is complex and works well only when the fraction of drifters is small compared to the rest of the network. Our routing protocol follows a different approach, in which the entire network uses a flat addressing scheme and does not require any subdivision into different areas or groups, nor does it require any mechanism for electing a "master" of each group.

The Cluster-based OLSR extensions [68] are another method proposed for a highly scalable adhoc routing protocol. It assumes some *clustering mechanism* is being executed in adhoc networks, and proposes to have OLSR operate at two hierarchical levels simultaneously, intra-cluster and inter-cluster. The intra-cluster OLSR traffic does not get forwarded beyond the limits of each cluster; only the inter-cluster traffic gets globally flooded, thereby reducing control traffic overhead. The authors offer no hint on exactly which clustering mechanism is to be used, how much time it takes to converge, or what happens to the formed clusters when the topology of the network changes radically. The simulation results for Cluster OLSR shown in [68] are based on 100 nodes, and some of those nodes are static and manually selected as cluster-heads. In contrast, our routing protocol does not require election mechanisms nor any kind of hierarchy; it achieves high scalability while remaining completely flat.

The IEEE 802.11s [26] is the standard for wireless mesh networks (WMN) using WLAN interfaces. In a 802.11s WMN, nodes perform the role of

Mesh Point (MP), which includes the exchange of routing messages and the frame forwarding. A node may also be a Mesh Access Point (MAP), which may require an additional standard 802.11 interface configured as an AP to provide access to legacy terminals; or/and a Mesh Portal Points (MPP), in which case it functions as a gateway to the Internet or to other non-mesh networks. The proposed routing for 802.11s is the Hybrid Wireless Mesh Protocol (HWMP), which is similar to AODV but operates at L2. It also includes tree-based routing for the WMN MPPs. Nodes initiate communication using the root-based tree node; this can create a bottleneck in the root node. Simultaneously, nodes broadcast a request for an optimized path between them, but this adds to the delay and can cause broadcast storms. User plane broadcast traffic, such as ARP and DHCP requests, exacerbates the broadcast storm problem. These facts contribute to the limited scalability of the 802.11s WMNs, which is known to support well only a few tens of nodes.

Most location-aided routing protocols (LAR [69], and derivatives such as PMLAR [70], and others [71, 72]) are reactive by nature and rely on the existence of location information (like GPS) in order to limit the flooding scope of route request messages. These routing protocols require a source node to be able to discover the location of a destination node. In some cases, a *location server* is employed, but clearly the communication with this server is an additional source of delay for route discovery (in addition to the route request / route reply pair), and limits scalability considerably due to its centralized architecture. In other cases, the position of a node is estimated based on the last known position, velocity vector, and the application of complex statistical models. The latter approach has two main problems. First, initially there is no "last position" known for any given node, and so the source node has to revert to classic flooding, which limits scalability. Second, the statistics involved are computationally intensive, and can potentially become a bottleneck, depending on router processing power and size of network; a router to put inside a vehicle must be small and not very powerful, and the network tends to be large. Finally, the assumption that location information is always available all the time may not hold if we consider routing in vehicles that spend considerable time underground, as is the case of subway trains.

The MAMP [73] protocol adds support for localized mobility management in mesh networks, achieving low handoff delay through the use of multipath routing. However, although this solution is mostly network based, it requires some special signaling with the mobile terminals, and therefore does not work with legacy terminals.

MobiMESH [74] is a wireless mesh network organized in a core area, responsible for the mesh routing and mobility management, and in an access area that supports legacy 802.11 terminals. The routing is performed using the OLSR protocol, and therefore limited to its scalability. To signal terminal mobility events, the authors propose the use of OLSR HNA messages, which are flooded through the core, and will not scale in a scenario with thousands of mesh nodes.

In [75] the authors propose a novel network-based local mobility management scheme called "Ant", which requires only network-side changes and manages mobility transparently for legacy terminals. It achieves very good handover interruption times, and it is efficient for both intra-domain communications and access to the Internet. As a downside, the Ant scheme relies on a Location Server, which needs to be often contacted, including once for every mobile host handover. This scheme therefore does not scale to tens of thousands of mobile hosts.

The SMesh [76] network supports fast handover of legacy 802.11 terminals connected to a mesh network, by allowing more than one access point to serve a wireless client. They both monitor the link quality, and the best link is selected. Unfortunately this requires the clients to be configured in 802.11 ad-hoc mode, which limits the available capacity of the network, due to requirement of a single-channel setup when in ad-hoc mode.

The Enhanced Mobility Management (EMM) proposal [77] for WMNs "manages mobility without the need, for end-users, to install any software or modify their protocol stack". It works with the Neighbor Discovery protocol of IPv6, which makes it an IPv6-specific solution. Additionally, it uses multicast request messages (MCREQ) to find out the location of terminals, which does not scale for large networks.

FastM [78] is an improvement of EMM that optimizes the case of handover of a terminal between two adjacent mesh routers. Due to the dynamic nature of the vehicular scenario, this mechanism may fail if handover occurs between two nodes that have not yet realized they are neighbors, for instance, a passenger switching from bus stop to a newly arrived bus.

The "Broadband Wireless Internet Access in Public Transportation" (BIT) project [79] addressed some of the problems described in this thesis, and confirms our approach of hetergeneous mesh networking, but did not propose any specific solution in detail.

3.8 Conclusions

In this chapter, new proposed architecture for the public transport vehicle networking scenario introduced in Chap. 1 was presented. The new architecture was named "WiMetroNet". This architecture entails two main contributions. First, the user plane that filters broadcasts and optimizes DHCP and ARP traffic via close integration of those protocols with the routing protocol, in contrast with 802.11s and TRILL which do not solve the broadcast issues. Another key difference is that frames are encapsulated using an MPLS header, allowing future protocol extensions without changing the data plane format, and enabling the solution to work also on non-IEEE 802 access links. Second, a new L2.5 routing protocol, that supports both mobile Rbridges and mobile end-user terminals, and feeds the data plane with IP/-MAC association tables, much needed for the DHCP/ARP optimizations; a feature that is also missing in 802.11s. The software architecture defined for the simulation/implementation is also described, in some detail, in this chapter.

We then presented routing optimizations to handle fast-handover of terminals in an efficient and scalable way, for large networks. This is something that is not available in either TRILL, 802.11s, AODV, or OLSR. In addition, a simulation-based evaluation of both this base solution and the mobility optimizations was presented. The new proposals have been demonstrated via simulation and analytical models, and the limits of scalability assessed for two different scenarios: a "road scenario", and a "city grid". For the "road scenario", we have shown that WMRP can scale to at least 2590 bus stops under conservative mobile 802.16 bandwidth estimates, using only 5% of bandwidth for signaling. It is more than enough to cover an entire city's worth of bus lines, which was our initial goal. In the "city grid" scenario, we show that the optimizations increase the covered area by a factor of 3.5 and the number of terminals increases by a factor of 12.25, when compared to a naïve link-state routing protocol, such as OLSR.

Chapter 4

Protocol development using ns-3

In general, researching and developing a new networking protocol involves several steps. These steps are depicted in Fig. 4.1:

1. We begin with a notion of a problem the protocol would solve in a given scenario. From the scenario/problem the researcher develops both a simulation model for the protocol and a simulation program that uses the protocol model;

2. Multiple simulations are run, with varying parameters;

3. The results are analyzed. As a result, the protocol model may be debugged, tweaked, or improved. The scenario simulation description may also need to be changed. As a result, we may need many iterations of simulation/analysis/improve;

4. When the simulation results are acceptable, it is time to stabilize the protocol model and write down the specification in a document;

5. From the specification, a protocol implementation is developed, possibly by a different team;

6. A testbed may be used to run the implemented protocol; trial runs may generate logs to be analyzed;

7. The logs are analyzed and if they report widely different results from what was expected from simulations then: a) implementation is buggy, b) the trial run exercises a slightly different scenario which has great

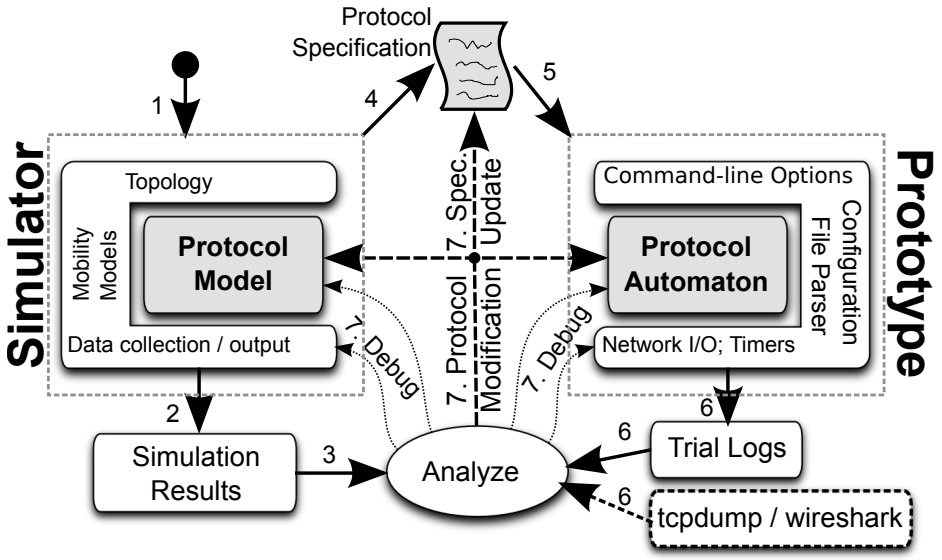


Figure 4.1: Traditional protocol development process

impact on the protocol performance. In case a), the implementation simply needs to be debugged, but in case b) the protocol model may need to be changed, new simulations run, and then the respective implementation changed accordingly, not forgetting the protocol written specification.

The development process, particularly using ns-3, contains some steps that could be optimized to make development easier and faster. In step 1, the researcher has to write a simulator which has essentially two parts: a model of a protocol and a simulation script. While the protocol model usually has to be written in C++, for performance reasons, writing the scripting part could be made much easier by switching to a high-level scripting language; this issue is addressed in Sec. 4.1. Additionally, writing the code for data collection is repetitive and tedious; Sec 4.2 proposes a generic data collection framework that automatically measures flows passing in a simulation with almost no effort required by the researcher to enable it. In step 3, the job of debugging a simulation program is very complex and time consuming, but the visualization tool proposed in Sec 4.3 can be of great value. Finally, the development process also has problems that become apparent when focusing on steps 5 and 7. First, there is a lot of duplicated effort when developing both a simulation model and an implementation prototype. Second, when

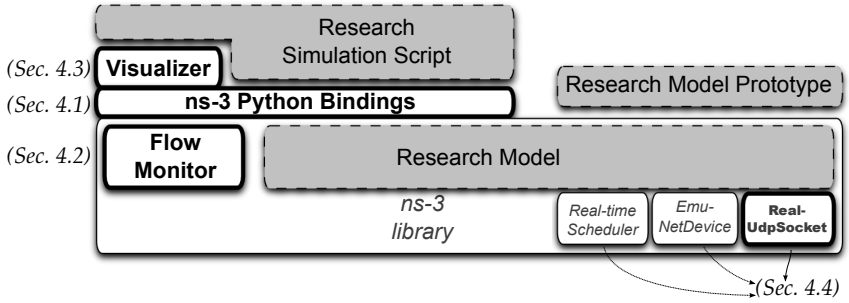


Figure 4.2: Ns-3 based protocol research framework, highlighting new contributions

modifications to the protocol are needed, after trial runs, they have to be done in three places at once: simulation model, protocol specification document, and implementation. The risk of inconsistencies between them being accidentally introduced is non-negligible. To address this issue, in Sec. 4.4 we propose modifications to development process.

Our attempt at solving these problems — a direct result of WiMetroNet research and development experience — involved creating a few ns-3 components, as well as measuring performance of other pre-existing components. As a result, a new research and development framework based on ns-3 emerges, illustrated by Fig. 4.2. In the figure, blocks named in italic represent pre-existing ns-3 modules, blocks in bold text represent ns-3 additions discussed here, and gray blocks represent code that needs to be developed by a R&D team specific for each research protocol.

4.1 Ns-3 scripting

While developing a new simulation model, or to evaluate an existing one, researchers have to make a program that builds the simulation scenario. This program involves creating a set of nodes, and for each node configure the network interfaces, applications that generate flows, and enable a mobility model to make the node appear to move as intended by the scenario description. This program is called the *simulation script*, because it is like a theater or movie script in the sense that it contains a list of actors (nodes and their applications) and actions that these actors must execute at specified times.

The simulation script can be written in any programming language. In ns-3, the only programming language initially available for simulation

scripts was C++. However, many simulators, including ns-2, allow simulation scripts to be written in a high-level dynamic language. The main advantages of using a high-level dynamically typed language are the following:

1. Programs tend to be shorter, more readable, and concise than statically typed languages such as C and C++. High-level data structures are often available at the language level and easier to use, and variables and parameters do not need associated type declarations. As a result, the programming job becomes easier and faster;

2. They allow for a much faster write/test/debug cycle of coding. Since there is no need for recompilation, after editing the source file, the script can be simply executed immediately. Because a program modification can be quickly tested, the run-time programming errors are caught quickly, and development occurs at a faster pace than with a C or C++ scripting program.

In this section, the ns-3 scripting interface is described. It is represented in Fig. 4.2 as the "ns-3 Python Bindings" component. First, we explain the ns-2 scripting interface and its main shortcomings, in Sec. 2.4.1. Then, in Sec. 4.1.1 we list the requirements for the ns-3 scripting interface that were considered. The PyBindGen tool that is the basis for the ns-3 python bindings is shortly described in Sec. 4.1.3, and its application to the ns-3 python bindings themselves presented in Sec. 4.1.4. The performance of PyBindGen is evaluated in 4.1.6, and finally some conclusions drawn in Sec. 4.1.7.

4.1.1 Ns-3 scripting requirements

The ns-3 scripting interface has been designed from the ground up to avoid the ns-2 scripting problems mentioned in Sec. 2.4.1. We have selected the programming language Python as the main scripting layer for ns-3, instead of TCL. This selection was based on multiple criteria: 1) language popularity and familiarity among developers, 2) readability, 3) breadth of language module library, 4) language performance. Regarding these criteria, we have found that Python is[1] the most popular of the dynamic languages if we discount PHP (PHP is rarely used outside the Web programming domain). Additionally, the Python language is designed with readability in mind; it

[1] According to "TIOBE Programming Community Index for January 2011"

even forces developers to indent their code properly by making indentation a part of the syntax. In terms of breadth of module library, Python is well served. Although Perl enjoys a wider module base, Python is said to come "with batteries included", meaning that it is distributed with a large library of modules useful to most applications. Performance-wise, the "The Computer Language Benchmarks Game"[2] places Python as one of the fastest scripting languages available, and with good perspective to become faster thanks to the Google-funded optimization project "Unladden Swallow"[3].

Having selected a programming language for ns-3, some design decisions had to be taken:

1. Python bindings for ns-3 have to be an optional layer on top of the C++ ns-3 library. In this way we avoid the run-time penalty of loading Python bindings in situations where they are not needed, for instance when deploying a protocol as a real implementation, as described in Sec. 4.4;

2. Python bindings should not create binding state for objects until they are effectively accessed from Python. In Python, each C++ object is usually represented in the Python run-time by another object that is called a "*wrapper*". Such a wrapper should be created for each object on demand;

3. Python bindings should have access to nearly the complete ns-3 API;

4. Maintaining Python bindings is tedious and error prone, especially if full API coverage is targeted. An automated solution to scanning the API is required;

5. It should, eventually, be possible to write new network models in Python, for quick prototyping purposes. This means that the Python bindings should allow the user to subclass an existing ns-3 class and override a virtual method, for instance;

4.1.2 Related work

Several tools exist to create Python bindings for C/C++ code. The main ones are:

[2] http://shootout.alioth.debian.org/
[3] http://code.google.com/p/unladen-swallow/

SWIG: SWIG (Simplified Wrapper and Interface Generator) is a tool that generates Python bindings from an interface description file. The main problems that SWIG has are the following: 1) it is written in C++, therefore is not possible to extend, 2) in order to support C++ classes it generates a pure Python module as a layer on top of a functional Python extension module, which is not very efficient, 3) the generated C code makes extensive use of preprocessor macros, making it difficult to follow and debug;

Boost-python: The Boost-python library is part of the Boost C++ project which aims to produce a set of C++ libraries to extend the functionality of C++. Writing Python extension modules with Boost-python is made difficult due to the very cryptic compilation error messages that are generated. Additionally, extension modules tend to be very large and a bit slow. Finally, extending Boost-python to handle strange cases can only be accomplished by experts in C++ template metaprogramming and Boost-python, due to the high complexity of the code base;

SIP: SIP is another generator for Python binding to C++ code that is written in C++. It generates efficient bindings, but it is not easily extensible;

Pyrex/Cython: The "pyrex" project was renamed to "Cython". It is a tool to generate Python bindings based on an interface description file. When we started developing the ns-3 Python bindings, Cython did not have support for C++ classes, so it was not considered a viable option.

None of the existing Python bindings tools was considered adequate. It has been clear from the start that, considering how ns-3 heavily extends the C++ type system with reference counted objects, custom smart pointers, callbacks objects, and object attributes, thanks to a heavy use of C++ templates, that any Python binding tool would need to be heavily customized and extended. However, most of the existing bindings tools are written in C++ and therefore are not easily extensible or modified. The case of Cython is different in that it is written in Python, but did not support C++, and is a highly complex system, being a complete compiler, not just a bindings generator, so it is equally difficult to extend to suit ns-3's purposes.

4.1.3 PyBindGen

Considering the limitations of the available Python bindings technologies, we developed a new Python binding tool from scratch, called PyBindGen (Python Bindings Generator). The main features of this new tool are the following:

1. Generates clean C or C++ code, almost as clean as what a human programmer would write. This is important to allow one to debug the generated binding code;

2. Generation is controlled exclusively by a Python API. This way we avoid the need to create a new interface definition language, and reduce the learning barrier. In addition, being a Python API, it makes it easier for PyBindGen to be integrated into a larger code generation framework, and designing it to be extensible via Python plugins is made simple;

3. PyBindGen has support for allowing errors to be logged and ignored, to make Python binding generation process more robust. This is important because, since the ns-3 API is very large and complex, and some methods that cannot be wrapped will inevitably appear, we allow them to be ignored instead of causing build errors;

4. PyBindGen generates self-contained python bindings, that do not require PyBindGen headers or library to be installed in the target system in which the bindings will be compiled or loaded. In addition, PyBindGen itself is a small pure-Python module, that can be included (and is included) in ns-3;

5. PyBindGen has support for most C++ features that ns-3 needs, including templated methods and classes, function/method overloading, virtual method re-implementation in Python, STL containers, reference counting, and smart pointers;

6. PyBindGen is able to scan C++ header files to find the API definitions nearly without developer intervention. For this, it uses GCC-XML, and the respective Python bindings, pygccxml.

Using PyBindGen to generate bindings for a C++ API is relatively simple. For instance, suppose we want to bind the C++ API defined by the following header:

```
#include <stdint.h>

class C
{
public:
  C ();
  C (uint32_t c);
  virtual ~C ();

  static void DoA (void);
  void DoB (void);
  void DoC (uint32_t c);
  uint32_t DoD (void);
  virtual void DoE (void);
private:
  uint32_t m_c;
};
```

The following Python script may be used to generate the respective Python bindings:

```
import sys
from pybindgen import retval, param, Module, FileCodeSink

mod = Module('c')
mod.add_include('"c.h"')

C = mod.add_class('C')
C.add_constructor([])
C.add_constructor([param('uint32_t', 'c')])
C.add_method('DoA', None, [], is_static=True)
C.add_method('DoB', None, [])
C.add_method('DoC', None, [param('uint32_t', 'c')])
C.add_method('DoD', retval('uint32_t'), [])
C.add_method('DoE', None, [], is_virtual=True)

mod.generate(FileCodeSink(sys.stdout))
```

4.1.4 Ns-3 Python bindings

In ns-3, PyBindGen is used in the following manner:

- PyBindGen is instructed to handle ns-3 smart pointers (class `ns3::Ptr`);
- A custom type handler for "`int argc, char *argv[]`" pairs of parameters is registered used to wrap `ns3::CommandLine::Parse`;
- ns-3 objects are wrapped with customized constructor wrapper to call `ns3::CompleteConstruct`, as required for ns-3 objects;

- Python bindings generation have two stages: first the API is scanned, periodically, using GCC-XML, and the result stored inside the ns-3 source code tree. For the other developers, the Python bindings can simply be generated using the API definitions already scanned and distributed, and they do not need to have GCC-XML installed;

- Hand-written wrappers for the `ns3::Simulator::Schedule` methods are included;

- As part of the API scanning process, a list of `Callback<>` template instantiations are recorded in a file, and during bindings code generation PyBindGen type handlers are registered, one for each callback type.

The resulting ns-3 Python extension module provides an API that is a simple and logical translation of the C++ API to Python. As an example consider the example in Listing 4.1, which simply creates a node with two sockets: one socket sends a packet to the localhost address via loopback interface, and the other socket has a "receive callback" attached that receives the packet. Both C++ and Python versions of the same simulation, where we can see that the Python one is a simple translation of programming language, albeit slightly smaller and more readable. Because the API is basically the same in either C++ or Python, there is no additional learning curve for developers to switch their programming language.

4.1.5 PyBindGen implementation approach

Some of the core PyBindGen classes are represented in Fig. 4.3. One of the basic building blocks is the **CodeSink** class, which is a container for lines of generated C code. It has methods to write lines of code, and automatically indents the code; the methods `indent()` and `unindent()` can be called to increase or decrease the amount of indentation. The class is abstract and is implemented by two subclasses: **FileCodeSink** writes the lines of code to a file, while **MemoryCodeSink** just records them in memory for later retrieval. The class **DeclarationsScope** keeps track of variable declarations. It is useful for when we want to declare a new C variable but want to make sure that our chosen variable name does not conflict with another variable with same name already used in the same scope.

The class **CodeBlock** represents a block of C code, in other words a code region. Its main function is to receive code lines, like CodeSink, but it additionally contains a DeclarationsScope, and can generate code to

Listing 4.1 A simple simulation script in C++ and equivalent Python

C++:

```
#include <iostream>
#include "ns3/core-module.h"
#include "ns3/helper-module.h"
#include "ns3/node-module.h"
#include "ns3/simulator-module.h"

using namespace ns3;

Ptr<Packet> _received_packet = NULL;

// Callback to read the packet
void _rx_callback (Ptr<Socket> socket)
{
  _received_packet = socket->Recv ();
}

int main (int argc, char *argv[])
{
  // Create an internet-enabled node
  Ptr<Node> node = CreateObject<Node> ();
  InternetStackHelper internet;
  internet.Install (node);

  // Create a socket to receive the UDP packet
  Ptr<Socket> sink = Socket::CreateSocket (node,
      TypeId::LookupByName ("ns3::UdpSocketFactory"
          ));

  sink->Bind (InetSocketAddress (
                Ipv4Address::GetAny (), 80));

  sink->SetRecvCallback (
        MakeCallback (&_rx_callback));

  // Create a socket to send a UDP packet
  Ptr<Socket> source = Socket::CreateSocket (
      node, TypeId::LookupByName ("ns3::
          UdpSocketFactory"));

  // Send the packet to localhost port 80
  source->SendTo (Create<Packet> (19), 0,
      InetSocketAddress (Ipv4Address ("127.0.0.1"),
          80));

    // Process the simulator events
  Simulator::Run ();

  std::cout << "Received a packet with " <<
      _received_packet->GetSize ()
          << " bytes." << std::endl;

  Simulator::Destroy ();

  return 0;
}
```

Python:

```
import ns3

_received_packet = None

# Callback to read the packet
def rx_callback(socket):
    global _received_packet
    _received_packet = socket.Recv()

# Create an internet-enabled node
node = ns3.Node()
internet = ns3.InternetStackHelper()
internet.Install(node)

# Create a socket to receive the packet
sink = ns3.Socket.CreateSocket(
    node, ns3.TypeId.LookupByName("ns3::
        UdpSocketFactory"))

sink.Bind(ns3.InetSocketAddress(ns3.
    Ipv4Address.GetAny(), 80))

sink.SetRecvCallback(rx_callback)

# Create a socket to send a UDP packet
source = ns3.Socket.CreateSocket(
    node, ns3.TypeId.LookupByName("ns3::
        UdpSocketFactory"))

# Send the packet to localhost port 80
source.SendTo(ns3.Packet(19), 0,
              ns3.InetSocketAddress(ns3.
                  Ipv4Address("127.0.0.1"
                  ), 80))

# Process the simulator events
ns3.Simulator.Run()

print "Received a packet with %i bytes."
    % (_received_packet.GetSize(),)
```

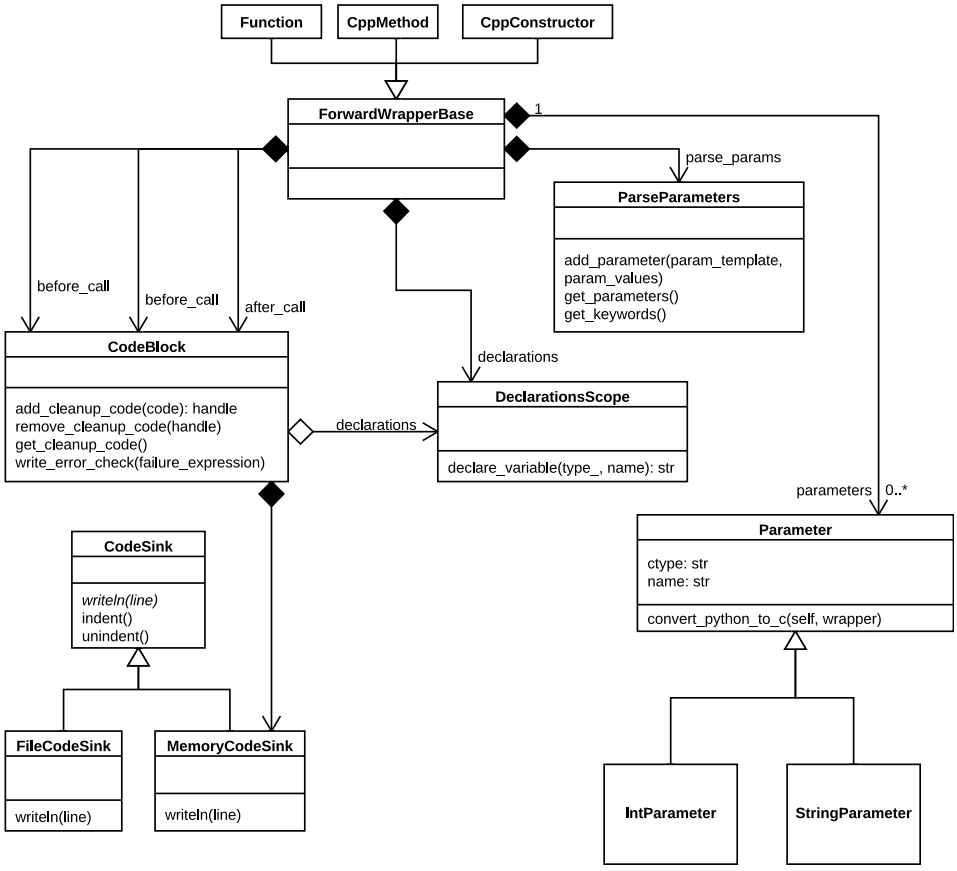


Figure 4.3: PyBindGen class diagram

handle error conditions. Handling error conditions in C is far from trivial because we have to keep track of all memory allocations that have been done previously and free all memory before returning. CodeBlock keeps track of "cleanup actions", which is C code to free memory that needs to be generated before returning from an error condition. In addition, the method `write_error_check` writes a C `if` statement that checks if a failure occurred and if so frees all memory and returns the error code.

Finally we have the main class for generating wrappers: **ForwardWrapperBase**. In Python bindings terminology, a "wrapper" is a C function that is invoked by the Python runtime, as a result of the corresponding Python function being called, and translates the python function call into the corresponding C function call. Python function wrappers invariably have the following general structure:

1. `PyObject* function_wrapper(PyObject *self, PyObject *args, PyObject *kwargs) {`
   Basically we declare that the wrapper returns a python object, and it receives three python objects:

   self − the instance of the method (this parameter is not used for function wrappers, only for constructors and methods of classes;

   args − this is a python tuple object containing the function parameters that are not expressed as named parameters;

   kwargs − this is a python dict object containing the function parameters that are expressed as named parameters;

2. "code before parsing parameters", e.g. initializing variables;

3. Parsing of parameters, via `PyArg_ParseTupleAndKeywords`;

4. "code before calling the C function", for instance additional parameter validation;

5. Call the C function;

6. "code after calling the C function", such as preparing the return value;

7. return the return value;

The **ForwardWrapperBase** class is designed to be extensible. To that end, it only provides a basic framework for generating the wrapper, but other parts of PyBindGen are responsible for actually supplying the code

for it to generate. Each of the code regions listed above is extensible: "code before parsing parameters", "code before calling the C function", and "code after calling the C function" are exposed as **CodeBlock** attributes of the wrapper, and the parameters used in the `PyArg_ParseTupleAndKeywords` are controlled by a **ParseParameters** class instance, also exposed as attribute. Most important for the code generation is the **Parameter** class, representing a single parameter for the function. The main properties of a function parameter are its C type and name. To generate code, **ForwardWrapperBase** calls the method `convert_python_to_c` of each of its parameters, passing itself as a "wrapper" parameter. Thus, the Parameter class implementation can add code to each of the **ForwardWrapperBase** extension points. In the end, **ForwardWrapperBase** only has to aggregate all the generated code and write it to the file in the correct order.

4.1.6 PyBindGen performance evaluation

To evaluate the performance of PyBindGen, we ran a series of microbenchmarks over a simple C++ API to be tested, shown in Listing 4.1.6. The benchmark tests are numbered 1 to 10, and consist of some Python code invoking operations over the C++ API using the Python bindings, repeating the operation a number of times, and measuring the time taken to complete. As a basis for comparison, we generated Python bindings using four tools: Boost.Python, SWIG, SIP, and PyBindGen. The benchmark tests are as follows:

1. The function "func1" is invoked with no arguments. The objective of this test is to measure performance of a simple function call;

2. The function "func2" is invoked with three literal floating point arguments. The objective of this test is to measure performance of a simple function call when function parameter parsing is involved;

3. The class "Multiplier" is instantiated, with no constructor arguments. The objective of this test is to measure the time taken to create a new instance of a wrapped C++ class;

4. The class "Multiplier" is instantiated, with a single float constructor argument. The objective of this test is to measure the time taken to create a new instance of a wrapped C++ class, but also to test the performance when constructor overloading (multiple constructors for the same class with different parameter types);

Listing 4.2 C++ API used in PyBindGen benchmark tests

```
void func1 (void);
double func2 (double x, double y, double z);

class Multiplier
{
    double m_factor;

public:
    Multiplier ();
    Multiplier (double factor);
    virtual ~Multiplier ();

    void SetFactor (double f);
    void SetFactor (void);
    double GetFactor () const;
    virtual double Multiply (double value) const;
};

double call_virtual_from_cpp (Multiplier const *obj, double value);
```

5. The method "GetFactor" of a preexisting "Multiplier" instance is called with no arguments. This test is to measure simple method call a C++ object;

6. The method "SetFactor" of a preexisting "Multiplier" instance is called with no arguments. This test is to measure simple method call a C++ object, with overloading involved;

7. The method "SetFactor" of a preexisting "Multiplier" instance is called with a single literal float argument. Again, tests the method call, but now with a different method call signature due to different passed parameters;

8. The method "Multiply" of a preexisting "Multiplier" instance is called with a single literal float argument. This tests the performance of calling a virtual method of a C++ class, which may be overridden in Python, but in this case is not;

9. The class "Multiplier" is subclassed, and the method "Multiply" is overridden in Python. The new implementation just chains to the base class implementation. An instance of the new Python-defined class is created, and then the test consists of calling the method "Multiply"

of this object with a single literal float parameter. This tests the performance of calling a virtual method of a C++ class, which in this case is overridden in Python;

10. The same object used in the previous test is used again to test calling a Python-overridden virtual method from C++. Instead of calling "Multiply" directly, we call a C++ function that calls this virtual method for us. This tests the performance of calling a virtual method of a C++ class, which in this case is overridden in Python. The difference from the previous case is that here the virtual method call is routed through C++, which takes a different performance penalty.

For each test, the operation is repeated ten million times, except in the last two tests that only repeat 2.5 million times. The results, in Fig. 4.4, are the execution time for each test and each tool relative to that of PyBindGen. For instance, we can see from the figure that test 5 (simple method call) takes nearly 3 times longer to complete in Boost.Python and SWIG than in PyBindGen. The SWIG tool is proved to be particularly bad at creating new objects, taking 7 or 25 times more time to create an object than PyBindGen, depending on the constructor invoked. PyBindGen is generally much faster than SWIG or Boost.Python, and is slightly faster in most tests than SIP.

These tests evidence a weakness of PyBindGen when dealing with overloaded constructors methods. The way PyBindGen handles overloading is by generating a different wrapper function for each different signature of the function, constructor, or method. Afterwards, an additional "master wrapper" is generated that calls each of the wrappers in sequence until one of them succeeds in parsing the parameters. Because multiple wrappers may be called with the incorrect signature, and for each tried incorrect signature a Python exception is generated, calling overloaded methods can take a lot of time. This is one area of PyBindGen that needs improvement, and which penalizes PyBindGen considerably when compared to SIP, for instance.

Another basis for comparison is the extension module file size, shown in Fig. 4.5. These sizes refer to the python extension module used above for the microbenchmarks, when compiled with optimization and no debugging symbols with GCC x86_64 on Ubuntu 9.10[4]. The Python module generated by PyBindGen is much smaller than the one generated by SWIG or Boost.Python. It is also slightly smaller than the one generated by SIP, in spite of SIP module not being standalone and requiring a SIP library to

[4]The software versions were: GCC 4.4.1, Python 2.6.4, SWIG 1.3.36, Boost.Python 1.38.1, and SIP 4.9.1-snapshot-20091015.

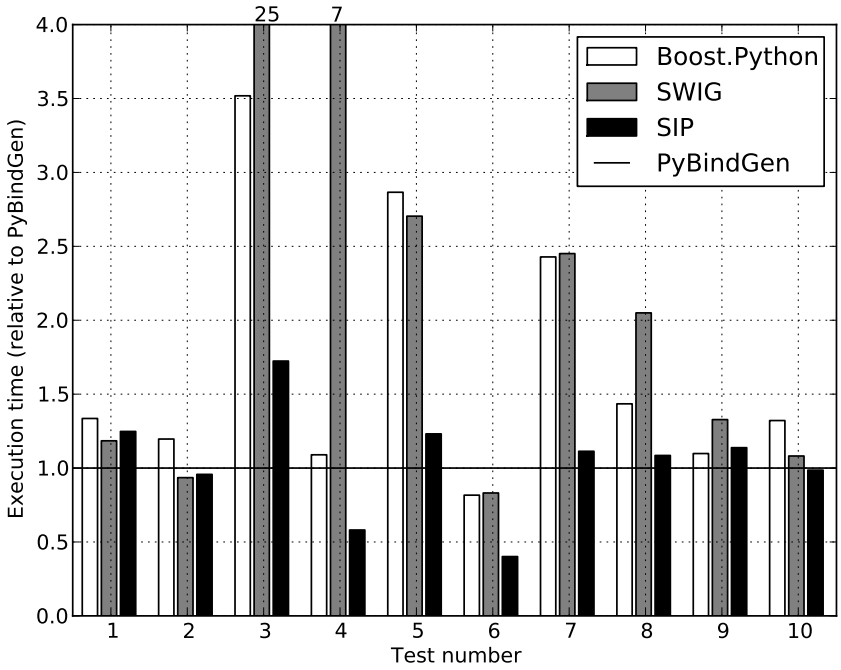


Figure 4.4: PyBindGen performance test results

be available in the system, in contrast with the PyBindGen module that requires no external library.

4.1.7 Summary

In this section we presented the motivation for providing a scripting interface for ns-3. Basically, a scripting interface is not a requirement to write simulations effectively, but is something that can speed up writing of simulation scenarios considerably. The ns-2 scripting interface, based on TCL, was explored and the main shortcomings identified. Some requirements for the ns-3 scripting interface were presented, and the "Python" programming language was found to be the best candidate to fulfill those requirements. Having selected a programming language, existing programming tools to bind C++ code to Python were discussed, and found to be suboptimal in a number of criteria.

As a result of our development efforts, a new tool called PyBindGen was developed to enable developers to bind C++ libraries to Python, and this tool was applied to ns-3 to produce in the current ns-3 Python bindings.

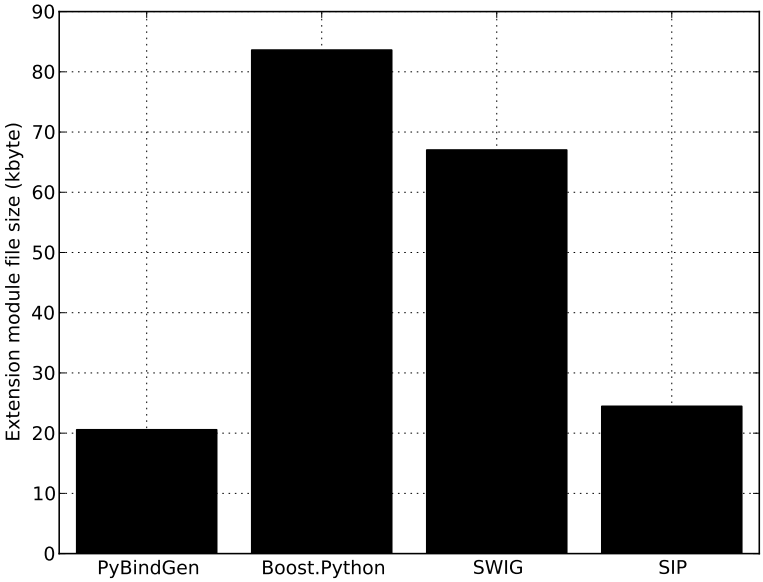


Figure 4.5: PyBindGen shared library size comparison

The ns-3 Python bindings are easy to maintain due to the automatic C++ header scanning ability of PyBindGen, cover most of the ns-3 C++ API, and do not deviate from the C++ API, making switching between the two languages relatively straightforward. The PyBindGen tool is then evaluated in terms of performance and is found to have much better performance than Boost.Python or SWIG. PyBindGen has similar performance to SIP, but is much more extensible via Python plugin code, and is more portable. Table 4.1 summarises the main differences found in the different tools.

Since a copy of PyBindGen is included in ns-3, and ns-3 has been downloaded thousands of times, it seems correct to assume that PyBindGen has been downloaded thousands of times by ns-3 researchers. It is also a successful open source project in its own right; the PyBindGen official released version 0.15 has been downloaded separately (not including the ns-3 downloads) 1148 times between the time of its release (August 15th 2010) and October 14th 2011.

4.2 Ns-3 flow monitor

Network monitoring is accomplished by inspecting the traffic in a network; this technique can be employed for purposes such as: fault detection — to

	PyBindGen	**Boost.Python**	**SWIG**	**SIP**
Extensible	Yes	Difficult	No	No
Gen. code readability	High	N/A	Low	Medium
Gen. code size	Low	High	High	Low
Performance	High	Low	Low	High
Gen. code has external dependencies	No	Yes	No	Yes
Automatic scanning of C++ headers	Yes	Yes (Py++)	No	No

Table 4.1: Comparison of Python bindings generators

detect disruptions in the network connectivity, routing or services; performance evaluation — to measure the network utilization or the performance of network protocols; security monitoring — to detect possible security problems; and Service Level Agreements (SLA) monitoring — to verify if a given network service is performing as specified in contract.

Network monitoring may pose the following problems: 1) monitoring strategy—it can follow a passive approach, where the traffic present in the network is measured, or an active approach, where test traffic is injected in the network for testing purposes. In both cases the data collected can then be sent to a central point where a complete picture of the network is computed; 2) monitoring points — not every network element can be easily monitored because nodes may lack a monitoring interface such as the Simple Network Monitoring Protocol (SNMP), or nodes may not support the installation of additional software. Monitoring points also depend on the test objective which can be monitoring network links, monitoring network queue dynamics, or monitoring the traffic generated by specific server applications; 3) monitoring duration — it must be large enough to enable the gathering of statistically sound results, what implies that a relevant number of events must be captured; this duration may be difficult to define in the passive approach, since the rate of relevant events is, a priori, unknown; 4) synchronization — we may be interested in monitoring a sequence of events that might be difficult to synchronize in scenarios involving several nodes; 5) transparency — because network monitoring often uses the same resources to transmit regular traffic and monitoring control traffic, we may say that monitoring may affect the results.

In network simulation environments, network monitoring is used mostly for characterizing the performance of network protocols. Monitoring in a

simulation environment differs from monitoring in real networks in a set of aspects: a) active monitoring is implicitly employed since the traffic injected in a simulated network is controlled and defined statistically; b) installing probing functions to inspect the traffic and queue dynamics is feasible and easy, and there is no need to use monitoring protocols such as SNMP, since the results are easily gathered as data resides in the same process; c) monitoring is less intrusive because the monitoring data needs not to traverse and use the network resources; d) events are easily synchronized because network simulators use the same simulation timer and scheduler; e) scenarios of lots of nodes can be easily addressed.

Network monitoring in simulated environments do present some problems: the simulation models may not be accurately designed and produce results that may diverge from the actual protocols. Gathering of results requires the researcher to develop a great deal of code and possibly to know and use different scripting languages. This may be aggravated by the unwillingness of the researcher to dedicate more effort to programming tasks, rather than focusing on the research. This may also lead to lower quality simulation models.

This section introduces the FlowMonitor, a network monitoring framework for the Network Simulator 3 (ns-3) which can be easily used to collect and store network performance data from a ns-3 simulation. The main goals behind the FlowMonitor development are to automate most of the tasks of dealing with results gathering in a simple way, to be easily extended, and to be efficient in the consumption of memory and CPU resources.

4.2.1 The FlowMonitor ns-3 module

Requirements

When designing the flow monitoring framework, FlowMonitor, a set of goals were taken into consideration, covering aspects such as usability, performance goals, and flexibility.

First, and foremost, the monitoring framework should be as easy to use as possible. Simulation is already very hard work, and researchers need to focus more on their research rather than spend time programming the simulator. The flow monitoring framework must be easy to activate with just a few lines of code. Ideally, the user should not have to configure any scenario-specific parameters, such as list of flows (e.g. via IP address and port src/dest tuples) that will be simulated, since these are likely to change for numerous reasons, including varying simulation script input parameters

and random variable seed. The list of flows to measure should itself be detected by the flow monitor in runtime, without programmer intervention, much like the existing "ascii" and "pcap" trace output functions do already in ns-3.

Another important concern is regarding the perfect amount of data to capture. Clearly, too little data can be risky for a researcher. A complex series of simulations, including varying input parameters, and multiple runs for generating good confidence intervals, can take between a few minutes to a few days to complete. It can be very frustrating to wait for simulation results for days only to discover in the end that there was *something* that we forgot to measure and which is important, causing the researcher to code in the additional parameter measurement and wait a few more days for new simulations to be run. Ideally, the simulation framework should attempt to include a reasonably complete information set, even though most of the information may not be needed most of the time, as long as it does not consume too much memory. A large data set is also useful because, in this way, the researcher is able to run the same simulations once, but analyze the results multiple times using multiple views. The reverse is also true. We do not want to save too much information regarding flows. Too much information is difficult to store and transmit, or can significantly increase the memory footprint of the simulation process. For instance, it is preferable to use histograms whenever possible rather than per-packet data, since per-packet data does not scale well with the simulation time. Histograms, on the other hand, do not grow significantly with the number of packets, only with the number of flows, while still being very useful for determining reasonably good approximations of many statistical parameters.

It is also a goal of this framework that the produced data can be easily processed in order to obtain the final results, such as plots and high-level statistics. As mentioned earlier, researchers usually need to analyze the same data multiple times for the same simulations, which means that this data should end up on a persistent storage medium eventually. Prime candidates for storage are 1) binary files (e.g. HDF), 2) ASCII traces, 3) XML files, and 4) SQL database. It is not completely clear which one of these storage mediums is the best, since each one has its drawbacks. Binary files, for instance, can store information efficiently and allow fast data reading, but are difficult to programmatically read/write, and difficult to extend to accommodate new information once the format has been defined, jeopardizing future extensibility. ASCII traces (line-by-line textual representation of data) are verbose (high formatting overhead), difficult to extend, and potentially slow to read. XML files have excellent extensibility traits, but are also

verbose, slow to read for large data sets, and require everything to be read into memory before any data filtering can be performed. SQL databases, on the other hand, are very efficient reading and filtering data without requiring much process memory, but can also be difficult to manage (except file embedded ones, like SQLite), difficult to extend with new information, and more difficult than textual files to find out how to read the data, since the data format can only be discovered by reading the documentation of the software that produced it or by using a database access GUI tool.

Initially, we have opted to support only XML for data storage output, as well as provide access to in-memory data structures. Since the FlowMonitor collects reasonably summarized information, it is not expected that XML trees will be very large, and reading such XML trees into memory is not a problem with today's computing resources. XML presents an important advantage over any other format, which is the large set of programming libraries for reading XML, for almost every programming language, and almost any platform. Especially in scripting languages, such as Python, reading XML is relatively straightforward, and requires no additional programming language, such as SQL. However, this issue is highly debatable, and so all the FlowMonitor data structures are made available for those who wish to serialize data into another storage format.

Support for Python based simulations is also one of the main design goals. The ns-3 Python bindings lack support for connecting callbacks to trace sources (`Config::Connect`, `Object::TraceConnect` and related APIs). Although supporting trace source callbacks in Python is desired and planned, the main reason for the lack of interest in implementing this feature stems from the awareness that allowing Python to do per-packet tracing operations would just slow down the simulations to the point of not being practical. The reasons for this slowdown include the need to, on a per-call basis, acquire the Python GIL (Global Interpreter Lock), convert the C++ parameters into Python format, call the Python code, convert the return values from Python into C++ format, and finally release the GIL. In order to effectively collect data for Python based simulations we need a "configure and forget" approach, wherein a C++ class is made responsible for the actual tracing and reports back to Python just the final results, at the end of the simulation.

The FlowMonitor architecture is also designed with extensibility in mind. One use case of simulation is to research next-generation networks, which may not even use IPv4, or IPv6. For instance, a researcher could be simulating an MPLS switching fabric, whose data plane encapsulates 802.3 frames, so we need extensibility at the packet acquisition level to accommodate dif-

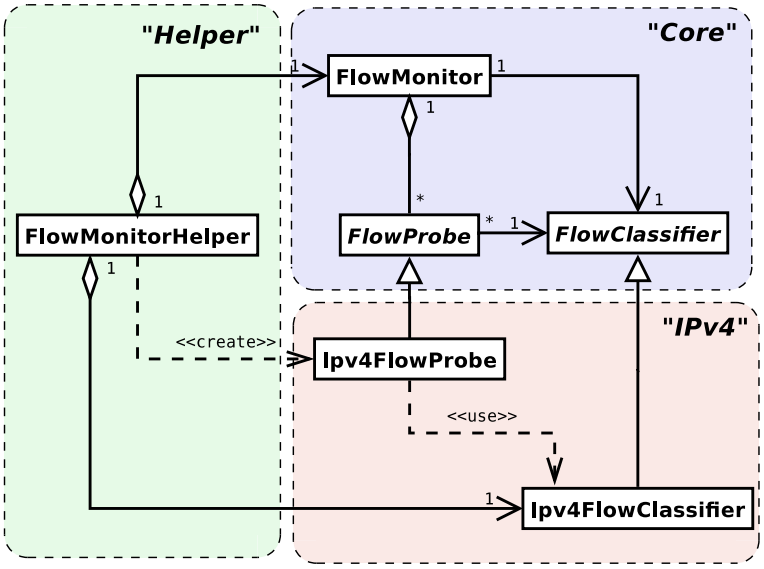


Figure 4.6: High level view of the FlowMonitor architecture

ferent data planes. Alternatively, the concept of "flow" may differ from the usual *five-tuple* (source-ip, dest-ip, protocol, source-port, dest-port). For instance, someone may want to classify flows by a "flow label" or DSCP IPv4 field. Therefore we also need an extensible flow classification mechanism.

Finally, there is the obvious, but ever so important, requirement of low monitoring overhead. While it is true that some memory and computation overhead cannot be avoided when monitoring flows, this overhead should be as low as can be reasonably expected.

Architecture Overview

The FlowMonitor module is organized into three groups of classes, as shown in Fig. 4.6. The group of "core" classes comprises the *FlowMonitor* class, *FlowProbe*, and *FlowClassifier*. The FlowMonitor class is responsible for coordinating efforts regarding probes, and collects end-to-end flow statistics. The FlowProbe class is responsible for listening for packet events in a specific point of the simulated space, report those events to the global FlowMonitor, and collect its own flow statistics regarding only the packets that pass through that probe. Finally, the class FlowClassifier provides a method to translate raw packet data into abstract "flow identifier" and "packet identifier" parameters. These identifiers are unsigned 32-bit integers that uniquely

identify a flow and a packet within that flow, respectively, for the whole simulation, regardless of the point in which the packet was captured. These abstract identifiers are used in the communication between FlowProbe and FlowMonitor, and all collected statistics reference only those abstract identifiers in order to keep the core architecture generic and not tied down to any particular flow capture method or classification system.

Another group of classes provides a "default" IPv4 flow monitoring implementation. The classes *Ipv4FlowProbe* and *Ipv4Classifier* subclass the abstract core base classes FlowProbe and FlowClassifier, respectively. Ipv4FlowClassifier classifies packets by looking at their IP and TCP/UDP headers. From these packet headers, a tuple (source-ip, destination-ip, protocol, source-port, destination-port) is created, and a unique flow identifier is assigned for each different tuple combination. For each node in the simulation, one instance of the class Ipv4FlowProbe is created to monitor that node. Ipv4FlowProbe accomplishes this by connecting callbacks to trace sources in the Ipv4L3Protocol interface of the node. Some improvements were made to these trace sources in order to support the flow monitoring framework.

Finally, there is also a "helper" group consisting of the single class FlowMonitorHelper, which is modeled in the usual fashion of existing ns-3 helper classes. This helper class is designed to make the most common case of monitoring IPv4 flows for a set of nodes extremely simple. It takes care of all the details of creating the single classifier, creating one Ipv4FlowProbe per node, and creating the FlowMonitor instance.

To summarize this high level architecture view, a single simulation will typically contain one FlowMonitorHelper instance, one FlowMonitor, one Ipv4FlowClassifier, and several Ipv4FlowProbes, one per Node. Probes capture packets, then ask the classifier to assign identifiers to each packet, and report to the global FlowMonitor abstract flow events, which are finally used for statistical data gathering.

Flow Data Structures

The main result of the flow monitoring process is the collection of flow statistics. They are kept in memory data structures, and can be retrieved via simple "getter" methods. As seen Fig. 4.7, there are two distinct flow statistics data structures, *FlowMonitor::FlowStats* and *FlowProbe::FlowStats*. The former contains complete end-to-end flow statistics, while the latter contains only a small subset of statistics and from the point of view of each probe.

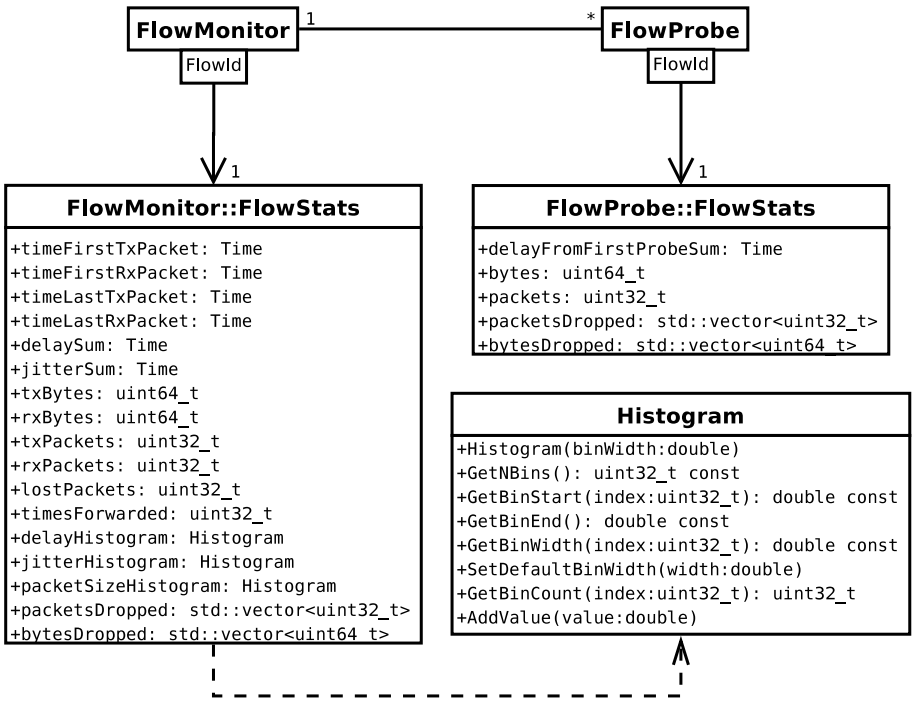


Figure 4.7: Data collected by the FlowMonitor

In order to understand the main utility of FlowProbe statistics, consider a simple three-node network, $A \leftrightarrow B \leftrightarrow C$. Consequently, we will have[5] three probes, P_A, P_B, and P_C. When a packet is transmitted from A to C passing through B, the probe P_A will notice the packet and create a FlowProbe::FlowStats structure for the flow, storing a $delayFromFirstProbeSum$ value of zero. Next, the probe P_B will detect the packet being forwarded, and will increment the value of $delayFromFirstProbeSum$ in its own FlowStats structure by the transmission delay from A to B. Finally, the packet arrives at C and will be detected by P_C, which then adds to its $delayFromFirstProbeSum$ the delay between A and C. In the end, we are able to extract not only end-to-end mean delay but also partial delays that the packet experiences along the path. This type of probe-specific information can be very helpful in ascertaining what part of the network is responsible for the majority of the delay, for instance. Such level of detail is missing from the FlowMonitor::FlowStats structure alone.

What follows is a more detailed description of the individual attributes in the flow data structures. In FlowMonitor::FlowStats, the following attributes can be found:

timeFirstTxPacket Contains the absolute time when the first packet in the flow was transmitted, i.e. the time when the flow transmission starts;

timeLastTxPacket Contains the absolute time when the last packet in the flow was transmitted, i.e. the time when the flow transmission ends;

timeFirstRxPacket Contains the absolute time when the first packet in the flow was received by an end node, i.e. the time when the flow reception starts;

timeLastRxPacket Contains the absolute time when the last packet in the flow was received, i.e. the time when the flow reception ends;

delaySum Contains the sum of all end-to-end delays for all received packets of the flow;

jitterSum Contains the sum of all end-to-end delay jitter (delay variation) values for all received packets of the flow. Here we define *jitter* of

[5]Note: in the abstract base architecture it is not implied that there is one probe per node; however, for the sake of this example we will assume the IPv4 flow monitoring case, which does make such assumption.

a packet as the delay variation relatively to the last packet of the stream, i.e. $Jitter\{P_N\} = |Delay\{P_N\} - Delay\{P_{N-1}\}|$. This definition is in accordance with the Type-P-One-way-ipdv as defined in IETF RFC 3393;

txBytes, txPackets Total number of transmitted bytes and packets, respectively, for the flow;

rxBytes, rxPackets Total number of received bytes and packets, respectively, for the flow;

lostPackets Total number of packets that are assumed to be lost, i.e. those that were transmitted but have not been reportedly received or forwarded for a long time. By default, packets missing for a period of over 10 seconds are assumed to be lost, although this value can be easily configured in runtime;

timesForwarded Contains the number of times a packet has been reportedly forwarded, summed for all packets in the flow;

delayHistogram, jitterHistogram, packetSizeHistogram Histogram versions for the delay, jitter, and packet sizes, respectively;

packetsDropped, bytesDropped These attributes also track the number of lost packets and bytes, but discriminates the losses by a *reason code*. This reason code is usually an enumeration defined by the concrete FlowProbe class, and for each reason code there may be a vector entry indexed by that code and whose value is the number of packets or bytes lost due to this reason. For instance, in the Ipv4FlowProbe case the following reasons are currently defined: `DROP_NO_ROUTE` (no IPv4 route found for a packet), `DROP_TTL_EXPIRE` (a packet was dropped due to an IPv4 TTL field decremented and reaching zero), and `DROP_BAD_CHECKSUM` (a packet had bad IPv4 header checksum and had to be dropped).

Some interesting metrics can be derived from the above attributes. For instance:

mean delay: $\overline{delay} = \frac{delaySum}{rxPackets}$

mean jitter: $\overline{jitter} = \frac{jitterSum}{rxPackets-1}$

mean transmitted packet size (byte): $\overline{S_{tx}} = \frac{txBytes}{txPackets}$

mean received packet size (byte): $\overline{S_{rx}} = \frac{rxBytes}{rxPackets}$

mean transmitted bitrate (bit/s):

$$\overline{B_{tx}} = \frac{8 \cdot txBytes}{timeLastTxPacket - timeFirstTxPacket}$$

mean received bitrate (bit/s):

$$\overline{B_{rx}} = \frac{8 \cdot rxBytes}{timeLastRxPacket - timeFirstRxPacket}$$

mean hop count: $\overline{hopcount} = 1 + \frac{timesForwarded}{rxPackets}$

packet loss ratio: $q = \frac{lostPackets}{rxPackets+lostPackets}$

Some of the metrics, such as delay, jitter, and packet size, are too important to be summarized just by the sum, count and, indirectly, mean values. However, storing all individual samples for those metrics does not scale well and is too expensive in terms of memory/storage. Therefore, histograms are used instead, as a compromise solution that consumes limited memory but is rich enough to allow computation of reasonable approximations of important statistical properties. In FlowMonitor, the class Histogram is used to implement histograms. It offers a single method to count new samples, AddValue, a method to configure the bin width, SetBinWidth, and methods to retrieve the histogram data: GetNBins, GetBinStart, GetBinEnd, GetBinWidth, GetBinCount. From this data, estimated values for N (number of samples), μ (mean), and s (standard error) can be easily computed. From the equations found in [80] (Chapter 2), we can derive:

$$N = \sum_{i=0}^{M-1} H_i$$

$$\mu = \frac{1}{N} \sum_{i=0}^{M-1} C_i H_i$$

$$s^2 = \frac{1}{N-1} \sum_{i=0}^{M-1} (C_i - \mu)^2 H_i$$

In the above equations, M represents the number of bins, H_i represents the count of bin i, and C_i the center value of bin i.

In FlowProbe::FlowStats some additional attributes can be found on a per-probe/flow basis:

delayFromFirstProbeSum Tracks the sum of all delays the packet experienced since being reportedly transmitted. The value is always relative to the time the value was initially detected by the first probe;

bytes, packets number of bytes and packets, respectively, that this probe has detected belonging to the flow. No distinction is made here between first transmission, forwarding, and reception events;

bytesDropped, packetsDropped tracks bytes and packets lost qualified by reason code, similarly to the attributes with the same name in FlowMonitor::FlowStats.

Example

To demonstrate the programming interfaces for using FlowMonitor, we create a simple simulation scenario, illustrated in Fig. 4.8, with a grid topology of 3×3 WiFi adhoc nodes running the OLSR protocol. The nodes transmit CBR UDP flows with transmitted bitrate of 100 kbit/s (application data, excluding UDP/IP/MAC overheads), following a simple node numbering strategy: for i in 0..8, node N_i transmits to node N_{8-i}.

It is out of scope of this study to present the full example program source code[6]. Suffice to say that enabling the flow monitor is just the matter of replacing the line
`ns3.Simulator.Run()`, with something like this (using the Python language):

```
flowmon_helper = ns3.FlowMonitorHelper()
monitor = flowmon_helper.InstallAll()
monitor.SetAttribute("DelayBinWidth",
                     ns3.DoubleValue(0.001))
monitor.SetAttribute("JitterBinWidth",
                     ns3.DoubleValue(0.001))
monitor.SetAttribute("PacketSizeBinWidth",
                     ns3.DoubleValue(20))

ns3.Simulator.Run()

monitor.SerializeToXmlFile("results.xml", True, True)
```

What the above code does is:

1. Create a new FlowMonitorHelper object;

[6] The complete source code is available online, at `http://code.nsnam.org/gjc/ns-3-flowmon`

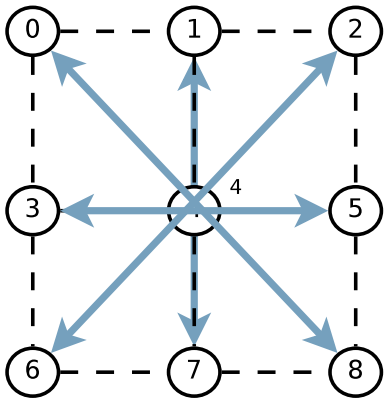


Figure 4.8: Flow Monitor example network topology

2. Call the method InstallAll on this object. As a result, the flow monitor will be created and configured to monitor IPv4 in all the simulation nodes;

3. Configure some histogram attributes;

4. Run the simulation, as before, calling ns3.Simulator.Run();

5. Finally, write the flow monitored results to a XML file named "results.xml". The second parameter of SerializeToXmlFile indicates if we wish to also save the histogram contents, and the third parameter indicates if we wish to save per-probe flow statistics.

To present the results, one would have to write a program that reads the data from XML and creates plots. For instance, the Python program in List. 4.3 reads the XML file and plots the histograms of 1) received bitrates, 2) packet losses, and 3) delays, for all flows. The program uses the Python XML parsing module *ElementTree* for reading the XML file, and the *matplotlib* module for generating the plots. The results in Fig. 4.9 are obtained from this, and show that most flows achieved an actual throughput of around 105 kbit/s, except for one flow that only transferred less than 86 kbit/s. Packet losses were generally low except for two flows. Finally, mean delays vary between $\approx$ 20 ms and $\approx$ 70 ms.

4.2.2 Validation and Results

For validation and obtaining results, we begin by describing a simple network scenario, which is then used for validation purposes. Finally performance is

Listing 4.3 Sample script to read and plot the results

```
et = ElementTree.parse(sys.argv[1])
bitrates = []
losses = []
delays = []
for flow in et.findall("FlowStats/Flow"):
    # filter out OLSR
    for tpl in et.findall("Ipv4FlowClassifier/Flow"):
        if tpl.get('flowId') == flow.get('flowId'):
            break
    if tpl.get("destinationPort") == '698':
        continue

    losses.append(int(flow.get('lostPackets')))

    rxPackets = int(flow.get('rxPackets'))
    if rxPackets == 0:
        bitrates.append(0)
    else:
        t0 = long(flow.get('timeFirstRxPacket')[:-2])
        t1 = long(flow.get('timeLastRxPacket')[:-2])
        duration = (t1 - t0)*1e-9
        bitrates.append(8*long(flow.get('rxBytes'))
                        / duration * 1e-3)

        delays.append(float(flow.get('delaySum')[:-2])
                      * 1e-9 / rxPackets)

pylab.subplot(311)
pylab.hist(bitrates, bins=40)
pylab.xlabel("Flow bitrate (bit/s)")
pylab.ylabel("Number of flows")

pylab.subplot(312)
pylab.hist(losses, bins=40)
pylab.xlabel("Number of lost packets")
pylab.ylabel("Number of flows")

pylab.subplot(313)
pylab.hist(delays, bins=10)
pylab.xlabel("Delay (s)")
pylab.ylabel("Number of flows")

pylab.subplots_adjust(hspace=0.4)
pylab.savefig("results.pdf")
```

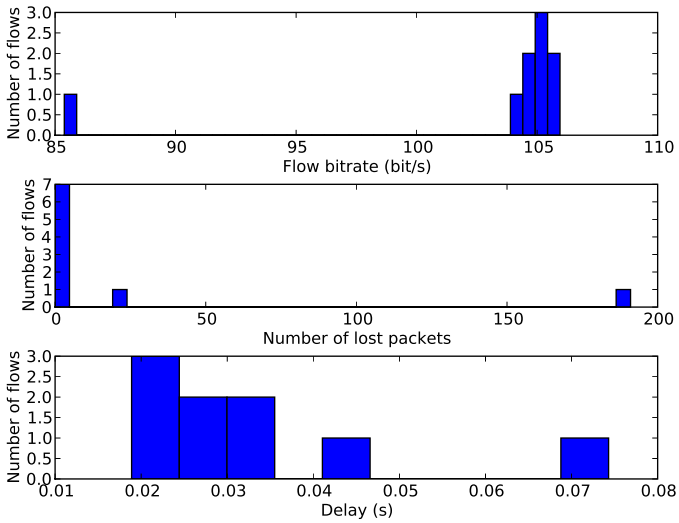


Figure 4.9: Flow Monitor example program results

evaluated using the same scenario and varying the network size.

4.2.3 Test scenario

Fig. 4.10 shows the network topology and flow configuration for the test scenario. It consists of a number of rows of nodes, each row containing a number of nodes, with each node connected to the next one via a point-to-point link. From left to right, a link is configured with 100 kbit/s, then the next link is configured with 50 kbit/s, the next with 100 kbit/s again, and so on. The configured delay is zero, and the drop tail queue maximum size

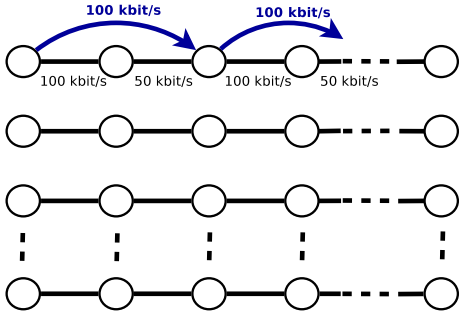


Figure 4.10: Flow Monitor test scenario

is 100 packets. The rows are not vertically connected. The ns-3 GlobalRoutingManager is used to compute the routing tables at the beginning of the simulation. Flows are created at the beginning of the simulation. Every other node sends one 100 kbit/s UDP flow[7] to the node that is two hops to the right, and sends another flow to the node that is two hops to the left. Packet size is the default 512 byte UDP payload.

Validation

To validate the flow monitor, we begin by examining the test scenario and deriving some theoretical flow metrics, which are then compared to values obtained by the measurements done with the help of the FlowMonitor module.

The test topology is relatively simple to analyze. In the first link, between the first and second nodes, we have one 100 kbit/s UDP flow in each direction. For each flow, the first hop of the flow traverses a link whose capacity matches exactly the flow link-layer bitrate. Thus, packet loss rate will be zero, and queueing delay will also be null. However, when the flow reaches the second hop it will have to be transmitted by a link that is half the capacity of the flow bitrate. Consequently, after a few seconds the drop-tail queue will fill to maximum capacity, causing a queueing delay, and half the packets will have to be dropped due to the bottleneck link.

We can derive the estimated values for delays, losses, and bitrates of each flow. We define S as the number of bits in each packet, $S = (512 + 20 + 8 + 2) \times 8 = 4336$ bit, and C_1 the bitrate of the link layer of the 100 kbit/s link, $C_1 = 100000$ bit/s. Then, the delay in the first hop, where there is no queueing, is simply the transmission delay, $d_1 = \frac{S}{C_1} = 0.04336$ s. In steady state, the second hop drop-tail queue will be filled, and so packets will experience a delay corresponding to transmitting 99 packets ahead in the queue, plus the packet itself, plus the packet that the PPP device is currently transmitting. Since the second hop has lower bitrate, $C_2 = 50000$ bit/s, and so $d_2 = 101 \times \frac{S}{C_2} = 8.75872$ s. Thus, the total end-to-end delay experienced by the flow will be $d_1 + d_2 = 8.80208$ s. Regarding packet losses, each flow traverses two hops. As previously explained, packet drop probabilities will be 0 and 0.5 for the first and second hop, respectively. Thus, the end-to-end packet probability will be 0.5 for each flow. Consequently, the received bitrate should be half the transmitted bitrate.

[7] Actually, at application layer the flow bitrates are 94.465 kbit/s, so that with UDP, IPv4, and MAC header overhead the bitrate is exactly 100 kbit/s at link layer.

Metric	Measured Value (95% C. I.)	Expected Value	Mean Error
Tx. bitrate	$99646.06 \pm 2.68 \times 10^{-5}$	99631.00	+0.015 %
Rx. bitrate	$49832.11 \pm 7.83 \times 10^{-5}$	49815.50	+0.033 %
Delays	$8.8005 \pm 8.8 \times 10^{-9}$	8.80208	-0.018 %
Losses	$0.4978 \pm 1.5 \times 10^{-6}$	0.5000	-0.44 %

Table 4.2: Validation results

The validation results in Tab. 4.2 show that, for a scenario with 2704 nodes (i.e. 1352 flows), the measured results (for 10 simulations) match the theoretical values within a very small margin of error. The expected values for transmitted bitrate is slightly less than 100 kbit/s due to the translation from layer-2 to layer-3, taking into account the factor $\frac{512+20+8}{512+20+8+2}$ due to the PPP header not being present where packets are monitored by the FlowMonitor. Same reasoning applies to the received bitrate. The errors found between estimated and measured values are negligible for transmitted/received bitrates and delays, and are likely a result of sampling issues and/or numeric errors. The error in the packet loss ratio is slightly larger, but this error is not of great significance. This error can be explained by the method of measuring losses used by the FlowMonitor, wherein a packet is considered lost if it has not been reported by any probe to have been received or retransmitted for a certain period of time, i.e. only packets that are "missing in action" are considered lost. Naturally, at the end of simulation a number of packets are still in transit, some of which could have been lost, but the potential packet losses are not accounted for, hence the error. In Chapter 5 we mention a possible way to overcome this error, as future work.

4.2.4 Performance Results

To evaluate the overhead introduced by flow monitoring, we ran a series of simulations, increasing the network size between 16 and 2704 nodes, and measuring the time taken to simulate each scenario, and memory consumption (virtual memory size), 1) without collecting any results, 2) with flow monitoring, and 3) with ascii tracing to a file. We repeat each experiment 10 times with different random variable seeds. The performance results in Fig. 4.11 show the additional overhead, in terms of memory consumption and simulation wall-clock time, that is incurred by enabling the flow monitor or trace file. The top-left plot shows the total memory consumed by a simulation while varying the number of nodes. Three curves are shown: one represents the simulations without FlowMonitor or file tracing enabled,

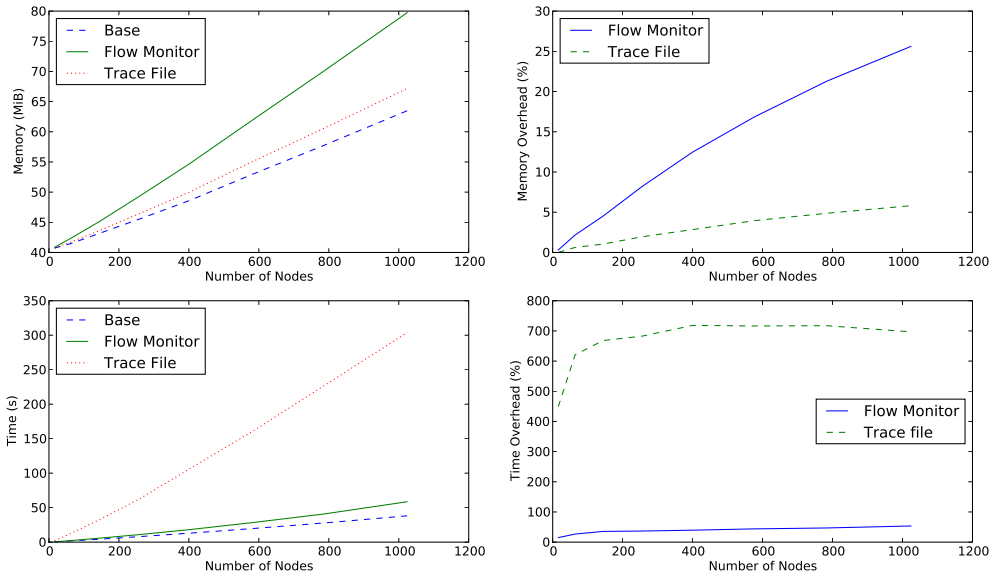


Figure 4.11: Performance results of the flow monitor

another other one represents the same simulations but with FlowMonitor enabled, and the remaining curve represents the simulations with ascii file tracing. The top-right plot shows an alternative view of the same information where instead the monitoring/tracing overhead is shown. The monitoring overhead, in percentage, is defined by the formula $100\frac{M_{monitor}-M_{base}}{M_{base}}$, where $M_{monitor}$ is the memory consumption with monitoring enabled, and M_{base} the memory without monitoring. Idem for file tracing. The memory overhead of enabling the FlowMonitor was 23.12% for 2704 nodes, corresponding to 45 MB of extra memory consumption, while the overhead with ascii file tracing was 5.82%. The bottom two plots show the impact in terms of simulation wall-clock time of the FlowMonitor and trace file, the left plot showing the three curves separately, the right plot showing the relative overhead (using a formula similar to the memory overhead one). For the simulation time, which in case of FlowMonitor includes the time needed to serialize all data to an XML file, the overhead reaches a peak of about 55% for 500 nodes, but then gradually decreases with the network size until reaching 38.82% (about 90 seconds) for 2704 nodes, while the overhead of ascii trace file generation is almost always above 700%. In the course of all simulations, over 150 GiB of ascii trace files were generated.

It should be noted that these values are worst case scenario. Since the

simulation scenario is very simple (PointToPointNetDevice is just about the simplest and fastest NetDevice implementation in ns-3), the additional overhead appears relatively significant. More complex simulations, for example WiFi and OLSR, should consume considerably more memory and CPU time, so that enabling the FlowMonitor on such simulations will add an overhead that will be smaller compared to the baseline simulation.

4.2.5 Related Work

There are a number of contributions that use trace files to gather information to produce network simulation statistics. **Tracegraph** [81] is a ns-2 trace file analyzer based on Matlab that is able to process almost every type of ns-2 trace file format; it produces a large amount of graphics that provide various views on the network such as throughput, delay, jitter and packet losses. In [82], a tracing framework is proposed, called **XAV**, which was built upon an XML Database Management System (DBMS). It was introduced to avoid storing redundant information, to speed up trace analysis and to simplify the access to trace files by post-processing tools. It requires simulators to produce XAV XML trace format files. These files are imported into the XML DBMS which provides an XQuery interface, an SQL-like querying mechanism. The authors compared the performance of XAV with an equivalent system using flat trace files and AWK for trace parsing, and the results show that XAV is always faster in situations where the user needs to extract non-multiple non-consecutive records.

The use of trace files can be optimized to a certain extent, but this approach usually requires a huge amount of disk space and the simulation performance is degraded by the extensive use of I/O operations. Even if sampling techniques are used, in large simulations the amount of data to store is still significant. Moreover, using samples introduces some precision error. In order to avoid the overhead of using trace files, statistics can be calculated during the simulation runs. In [83], an experiment was conducted which concluded that using trace files can make simulations take up to 6 or 7 times longer when compared to an approach where statistics are gathered during the simulation. The authors proposed a framework for ns-2 that integrates a data collection module with a separate graphical statistic analysis tool that reads result files produced by the former. The data collection module consists of a static C++ class named **Stat**. The Stat class provides a put method that can be called anywhere in the NS-2 code. It can collect abstract metrics, enabling the use of the framework to measure any variable needed by the user. It is also possible to select specific flows

	Tracegraph	**XAV**	**Stat**	**ns-2 traces**	**FlowMonitor**
Complexity	Low	High	High	Medium	Low
Performance	Low	Medium	Varies	Very low	High
Flexibility	Low	High	High	High	Moderate

Table 4.3: Comparison of simulation data collection frameworks

as targets for monitoring instead of considering all the flows. The collected data is used to calculate either mean values or probability density functions. The statistical analysis tool, named analyzer, uses a configuration file which defines the minimum and maximum number of simulation runs required and the specific metrics the user is interested in analyzing. The number of simulation runs can be defined by setting a desired confidence interval. The analyzer can be configured through command line or using a GUI developed with C++/GTK+. The generic nature of this framework provides enhanced flexibility, but at the cost of simplicity. It requires the programmer to explicitly include calls to Stat::put method in his code. Although the framework includes code to automate calls for the most common types of metrics, it still can require some integration effort in some cases. Its use is limited to ns-2. Porting the framework to ns-3 is not trivial due to the many differences between the two versions.

Table 4.3 summarises the presented data collection frameworks. **Tracegraph** is simple to use, but is relatively slow to process results[8], and is not easy to generate new types of data or plots. **XAV** is built around XML (including XPath) and XML based DBMS technologies, and creates DB-based traces of packet transmissions and receptions. Thus, researchers need to post-process these results, leading to complexity of the framework. Moreover, in spite of DBMS optimizations, the logging of per-packet information limits the the performance considerably. On the upside, the per-packet traces give the framework a lot of flexibility. The *Stat* framework is relatively complex for researchers, since Stat::put calls need to be added to simulations, but it is highly flexible. Its performance varies with the specific simulation: the more data is logged, the worse the performance will be. Regarding plain **ns-2 tracing**, the approach of generating detailed trace files offers great flexibility, but low performance and high complexity. Finally, the *ns-3 Flow Monitor* requires nearly no configuration, and it outputs most commonly used metrics in either both a simple C++ data structure

[8]Especially since it requires ns-2 trace files to be generated first.

or a XML file, whichever the researcher prefers. Since the data is highly summarized, Flow Monitor has very good performance. It features some flexibility, given its extensible architecture, allowing the researcher to create new probes and classifiers, but extending to new types of metrics may not be so simple: it measures flows, and nothing else. We may consider Flow Monitor a framework for measuring data for most simulations, but its usage can be complementary to more complex data gathering facilities.

4.2.6 Summary

In this section we have described a solution that solves a common problem for all researchers that need to conduct simulations in ns-3: how to easily extract flow metrics from arbitrary simulations? Existing solutions, some of which have been identified, do not solve this problem effectively, both for simulators in general but especially in ns-3. A set of general requirements have been identified, and a new flow monitoring solution was designed and implemented which meets those requirements. The simplicity of use of this new framework has been demonstrated via a very simple example. The implementation was validated by comparing measured flow metrics with theoretical results. Performance results show that flow monitoring introduces a relatively small overhead, even when used with a base simulation that is already very efficient to begin with. The Flow Monitor module has been a part of ns-3 since version 3.6 (October 2009).

4.3 Ns-3 visualizer

During the process of writing simulation models or simulation scripts, the developer is often faced with unexpected simulation problems when executing the simulation. These problems may stem from programming errors (so called "bugs"), or even from design errors. The programming errors, which may be located in either the protocol model or in the simulation script, may result in 1) memory access violations, such as segmentation fault, 2) assertion failures, or 3) invalid behavior/results. Errors in design can be located in the simulation script, resulting in a simulation that is not at all functional (e.g. a missing IPv4 route, or incorrect SSID configured in a WiFi NetDevice), or in the protocol model itself, resulting in an implementation of the protocol that is a correct software translation of the initial design but does not effectively solve the problem at hand.

Detecting, identifying, and correcting these problems can be a tedious process. It is not uncommon for researchers to spend more time correcting

simulation errors than writing the simulation model and script in the first place. The method and difficulty of correcting problems varies greatly with the type of problem.

Memory access violations (not to be confused with memory leaks) are usually the easiest of errors to detect and solve. Indeed, the OS automatically aborts the execution of the simulation process with an error message. Afterwards, a simple debugger can pinpoint the exact line source line where the memory violation occurs, and the call stack can help figure out the code responsible for the error. For more complex memory errors, additional tools, such as *valgrind*, may be be used.

Assertion failures may sometimes be simple to solve, or may be a symptom of a more difficult problem. They are placed inside the code to explicitly detect and report invalid states. An invalid state may be reached due to a trivial coding error, or due to a more complex sequence of events, which may denote a potential protocol design error. The former can be solved by running the simulation under a debugger and subsequent code review. The latter usually requires reproducing the problem with simulator logging enabled and then analyzing the logs carefully to identify the sequence of events that led the protocol state machine into the invalid state.

Finally, one needs to deal with errors that affect the simulation results, but otherwise do not have a clear "alert" signal to guide the researcher. Sometimes, the result is obviously wrong, but the underlying cause is not clear. For instance, a node may not be receiving a TCP flow from another node via a WiFi link, but there are several causes for this, such as: 1) the nodes are not in range of the WiFi, 2) the routing protocol may not have discovered a correct route when the application tries to establish a connection, 3) the receive socket is not bound to the correct address, or even 4) the WiFi interface is configured with an incorrect IPv4 address or network mask. Tracking down these errors usually involves enabling simulator logging and carefully checking that the intended node configuration is being effectively applied to all layers of the network stack. Occasionally, errors are more subtle, as they may affect the results only slightly. To detect these errors, carefully monitoring of the results during simulation is required. For instance, the bitrate of a flow between two wireless nodes may be lower than expected because an error in the mobility models keeps the nodes too distant for much of the simulation time.

A lot of the more complex debugging work will consist in enabling logging and analyzing log files. When developing a protocol implementation, pretty much the same technique is usually employed as with a simulator. However, while in the implementation case the developer only has to deal with log

messages of one, or a few, nodes, in the case of a simulator logging produced for many nodes in the simulation, making the job of simulation log analysis significantly more difficult.

One way to mitigate these problems is to develop some tool that would let developers visualize graphically the simulation state and its evolution over time. Such tool would allow developers to catch many mistakes very quickly by simple inspection of the visual representation of the simulation. The ns-3 "PyViz" visualizer tool is our attempt to do just that.

4.3.1 The "PyViz" ns-3 visualizer

The ns-3 visualizer, codenamed "PyViz", evolved gradually from two earlier attempts. The first visualization code submitted to ns-3 was a simple program that represented graphically an ns-3 mobility model, in October 2007[9]. Later, in February 2008[10], we produced a new version of it that instead of a single standalone program worked as a visualization library that could be applied to an existing simulation. Eventually it became clear that programming a visualization tool completely in C++ was rather tedious and would lead to very slow development pace. Work on a visualization tool mostly written in the Python programming language, started shortly after, and a first version of it was published in September 2008[11]. The project evolved and became highly popular, being used by the main ns-3 developers to demonstrate ns-3 in important simulation workshops, such as in the Aug. 2008 SIGCOMM workshop. However, due to potentially contending related work, namely iNSpect, and Animator, only recently our tool PyViz was merged into the main ns-3 tree, and released with ns version 3.10.

4.3.2 PyViz architecture

The PyViz visualizer was developed using a Python and Gtk+ based GUI programming stack, with GooCanvas library as the main simulation canvas framework, as shown in Fig. 4.12. While Gtk+ provides the normal application GUI elements, such as windows and buttons, GooCanvas provides a *retained mode* "scene graph" rendering library. While OpenGL is a popular library for these sort of visualization tasks, it provides more "low level" graphics drawing primitives. The GooCanvas library is more "high

[9] `http://mailman.isi.edu/pipermail/ns-developers/2007-October/003399.html`
[10] `http://mailman.isi.edu/pipermail/ns-developers/2008-February/003759.html`
[11] `http://mailman.isi.edu/pipermail/ns-developers/2008-September/004729.html`

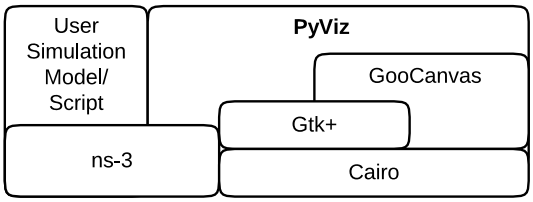


Figure 4.12: PyViz software stack

level" and allows for quicker development. The PyViz visualizer is written in Python and calls Gtk+, GooCanvas, and ns-3 APIs.

The PyViz visualizer internal work-flow is represented by the activity diagram in Fig. 4.13. When starting, the network topology is first scanned, and for each simulated node a corresponding canvas item is created to represent it. Then, the visualizer splits into two separate threads of execution. In one thread, the simulation is executed, in 100 ms steps, by command of the GUI. In the other thread, the main thread, the GUI is updated every 100 ms. Thus, the simulation and visualization threads advance in lock-step. Thanks to the dual-threaded architecture, it is possible for the ns-3 simulator to be simulating the next 100 ms while the visualization thread is free to process GUI events, and the GUI becomes more responsive as a result. Two native Python thread synchronization objects are used from the standard "threading" Python module. A `threading.Lock()` object is used to make sure that the visualization and simulation threads do not access the ns-3 API at the same time, which would otherwise cause memory corruption due the fact that ns-3 API is not generally thread-safe, with only a few exceptions. Additionally, a `threading.Event()` object is used to allow the visualization thread to send a "go" command to the simulation thread whenever it is time to simulate the next 100 ms chunk, but without having the simulation thread busy-wait when waiting for the command.

PyViz has a plugin architecture: it automatically loads plugin modules, as found in a certain directory, and executes the plugin code at startup. Then, plugins typically connect to a set of "signals" on a Visualizer object, such as:

populate-node-menu: this signal may be used to add options to the right-click popup menu of nodes;

simulation-periodic-update: this signal is emitted periodically as the simulation progresses;

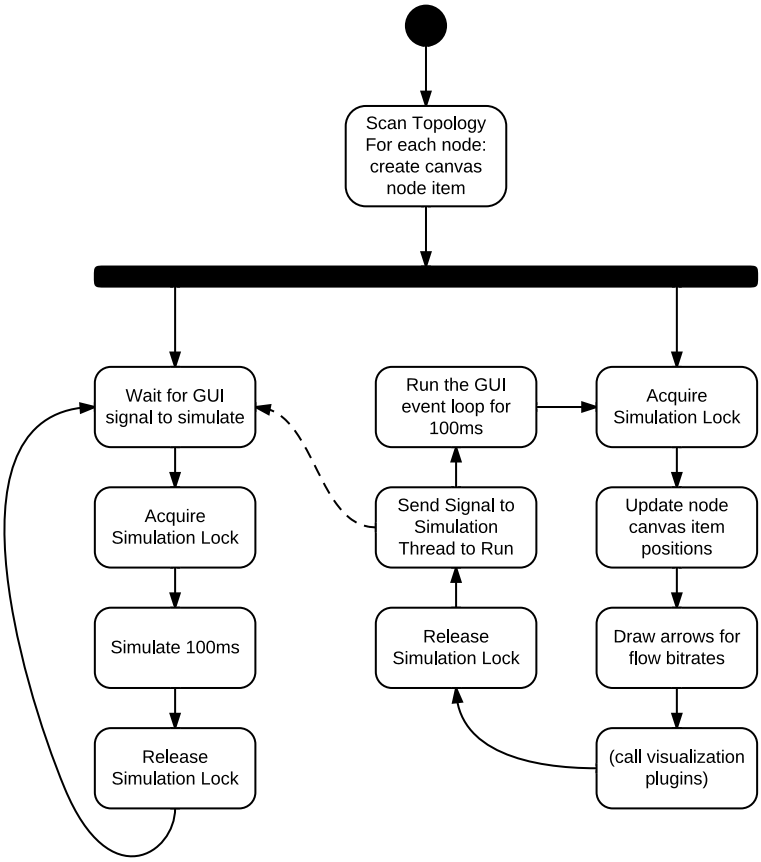


Figure 4.13: PyViz activity diagram

topology-scanned: this signal is emitted only once, right after the topology is scanned;

update-view: this signal is emitted periodically after the view is updated by pyviz, to allow plugins to add additional visualization elements.

4.3.3 PyViz features

Fig. 4.14 shows a screenshot of a typical visualization session with PyViz. Some of the features supported by PyViz include:

- Both wireless and wired links are supported;
- There is an API to change the color of individual nodes, or even assign an SVG icon;
- The simulation may be paused and resumed at any time;
- There is a control to speed up or slow down the simulation;
- It is possible to drag a node in the visualizer, while the simulation is running, the the new position having effect on the simulation results. This way, it is possible to quickly test the impact of certain mobility patterns on the simulation model. For example, we can test the range of a wifi link by dragging a node away from the access point until the connection is lost, or we can test the impact of handover on a routing protocol state;
- It is possible to open a console window that allows one to execute Python commands. These commands can do anything that the ns-3 API provides. They can not only query the simulation state, but also change the simulation in many ways, for "quick-and-dirty" experiments;
- The visualizer displays bitrate of traffic exchanged by nodes, in a way that scales well to any amount of data exchanged between nodes, since no individual packets are displayed, only the aggregate bitrate of all packets exchanged between each pair of nodes;
- Placing the mouse pointer over a node displays a "tooltip" with some useful information about the node, as shown in Fig. 4.14. Plugins may even add additional information to this tooltip;

- PyViz automatically proposes a layout for nodes that do not have any position defined in the simulation[12], using the graphviz [84] library for this potentially complex layout task.

Some useful plugins are included in PyViz:

interface_statistics.py: adds a popup menu item for nodes, with an option to display an information window containing a list of interfaces and transmitted/received bitrates for each interface;

ipv4_routing_table.py: Adds a node option to display the contents of the static IPv4 routing table in an information window; this information is updated dynamically as the simulation progresses. This is seen in Fig. 4.15;

olsr.py: idem, but for the OLSR routing table;

show_last_packets.py: Adds an option to display a window with a list of packets transmitted, received, and dropped by the node;

wifi_intrastructure_link.py: detects nodes operating in WiFi infrastructure mode — access points and stations — and displays a dashed red arrow between each station and the access point it is associated with.

4.3.4 Related Work

For visualization, the best known example is the official ns-2 visualization tool, "Nam". It is written in Tcl/Tk, and is a "post-mortem" visualizer. It reads a tracefile that is generated by ns-2 describing, line by line, the positions of nodes at each instant, as well as packet transmissions that may occur. In this respect, Nam is essentially a player of animations, not an animator of a running simulation. That is main difference to PyViz: while PyViz animates a running simulation, Nam animates a simulation that is already finished. With the Nam approach of reading trace files, visualization is limited to the information provided in the trace file, while PyViz can take any kind of information provided by the ns-3 API. Another limitation of the post-mortem approach is that it is not possible for the animator to affect the simulation in any way, like PyViz does when dragging nodes with the mouse pointer.

[12] Typically, wired nodes do not have position set because they do not need it for simulation purposes.

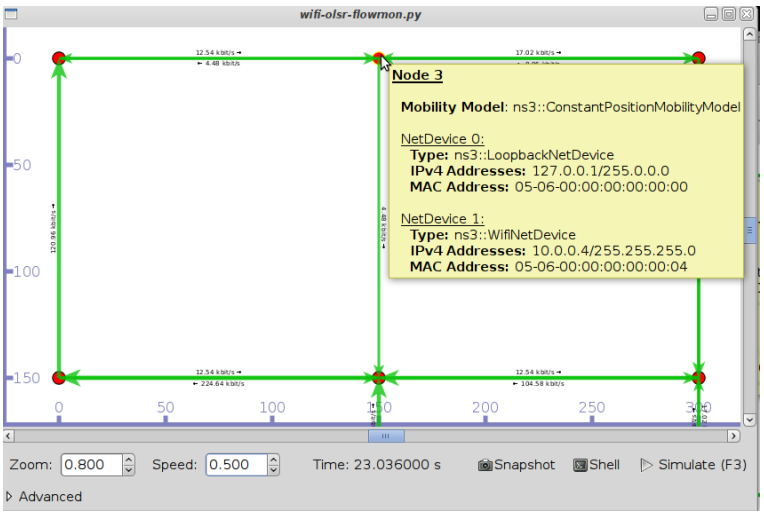


Figure 4.14: Screenshot of PyViz in action, with tooltip over a node

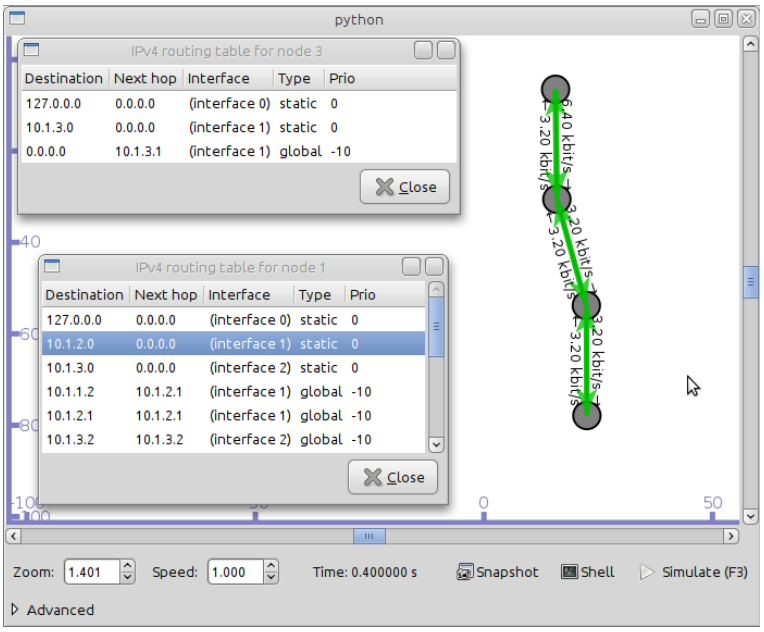


Figure 4.15: PyViz showing routing tables of two nodes

	Nam	Tkenv	iNSpect	NetAnim	PyViz
Simulator	ns-2	OMNET++	ns-2[a]	ns-3	ns-3
Automatic layout of wired nodes	Yes	Yes	Yes	No	Yes
Live interaction w/ simulation	No	Yes	No	No	Yes
Sim. script modifications needed	Minimal	None	Minimal	Some	None
Scalability	Good	Moderate	Excellent	Good	Moderate

[a] Experimental support for ns-3 started, but does not appear to be actively developed anymore.

Table 4.4: Comparison of simulator visualization tools

Within OMNET++, the Tkenv execution environment runs the simulation while it is visualized graphically within a Tcl/Tk interface. It is a very mature graphical environment, compared to PyViz. It allows changing some state of the simulation interactively, but there is no possibility of dragging a node with the mouse to change its position.

Another popular example, initially for ns-2 but now also ported to ns-3, is iNSpect [85]. It uses the OpenGL library to provide very fast graphical rendering with hardware acceleration. However, it is also tracefile based, in addition to be written in C++, which limits the ability to extend it or provide plugins. Nonetheless, the idea of iNSpect of using arrows to represent flow bitrates, instead of Nam's approach of drawing individual packets, was inspirational to PyViz.

Specifically for ns-3, the NetAnim animator was contributed by George Riley (Georgia Tech), but it is written in C++, and is tracefile based.

Table 4.4 summarises some of the differences between the visualization tools analyzed. Regarding simulator support, Nam and iNSpect support ns-2, while NetAnim and PyViz are made for ns-3 instead. The iNSpect team actually started some work for supporting ns-3, but this work appears to be unfinished; the iNSpect home page[13] does not even mention ns-3 at all. All the visualization tools except NetAnim support some form of automatic node layout algorithm; this enables them to automatically assign "reasonable" positions for nodes which do not have a mobility model or coordinates set in the simulation. PyViz is the only one that allows live interaction with the simulation, allowing the researcher to pause the simulation and manually change some state in the middle of it, for instance change a node's

[13] `http://toilers.mines.edu/Public/Code/Nsinspect.html`

position or stopping an application. The other tools are "post-mortem" visualization tools, which means that they can only run once the simulation is finished, and therefore are not allowed to modify the simulation state in any way. To enable visualization in a given simulation, Nam and iNSpect require only minimal modifications to the simulation: typically just a couple of lines to enable tracing in the visualization format. NetAnim requires some modifications in the case of wired nodes, namely to add a constant position mobility model to such nodes. With respect to scalability, meaning how well the visualizer supports networks with many nodes and many packets, the visualization tools that use trace files have some advantage, since when visualizing a simulation the simulation has already been run, therefore all computing resources can be devoted to the graphical interface. PyViz and Tkenv do not scale so well to networks with many nodes, since the visualization happens at the same time that the simulation runs. However, typically a well written network simulation scenario is parametrized in such a way as to allow reducing the number of nodes via command-line option. Thus, the researcher can reduce the number of nodes just for visualization purposes, in order to debug the problem graphically, and then go back to a large network scenario with graphics turned off.

4.3.5 Summary

Debugging simulations is a potentially daunting task, or at least extremely time consuming for experienced researchers. We have presented in this section a visualization tool for ns-3 that allows the researcher to quickly find some simple mistakes, thereby saving significant development time, while giving more confidence that the simulation is behaving as intended. The visualization tool is shown to have very unique features, not found in any other visualization tool, derived from its "live" visualization approach. These unique features include the ability to drag simulated nodes during simulation, the ability to inspect any simulation data via a Python interactive console, ability to modify the simulation state via the same Python console, and the plugin architecture allowing the visualizer to be extended in Python.

4.4 Fast prototyping of protocols using ns-3

Research and development today has to keep up with a fast evolving field of research, and communications protocols research is no exception. Considerable time is spent by researchers and developers from the idea of a protocol that solves a specific problem and the deployment of an implementation of

that protocol. One of the main contributors for the time spent is the need to develop both a simulation model and then an implementation of the same protocol. What if we could develop a single hybrid simulation/implementation model? In this section we explore the possibility of doing just that: using the Network Simulator 3 (ns-3) as a framework for developing new protocols that can be first simulated and then deployed in a real communications device with only minimal changes. Such an approach appears to be interesting, at the surface, since it will save development time, among other advantages. However, questions regarding real world performance remain, and one of the objectives of this thesis is to discover performance limits of the proposed framework.

This section describes three main contributions. First, the proposal of a new unified simulation/implementation protocol development process that takes advantage of the existing ns-3 network emulation functionality. We additionally evaluate the packet processing performance, in terms of achievable throughput, packet loss, and round-trip time, of ns-3 working in emulation mode, compared to a pure kernelspace IPv4 forwarding. Finally, we propose additional ns-3 classes that improve the performance of emulation of control plane protocols, and simplify the deployment of such protocols.

4.4.1 Proposed improved protocol development process

One has to wonder what is so different between simulation model and implementation to warrant duplicated code. A network protocol can generally be described as a *Timed Automata* [86], i.e. a state machine in which state transitions are triggered by input messages and constrained by the passage of real time. Both simulation and implementation of the same protocol include the very same automata, only the way messages are received and transmitted, as well as the way time passage is measured, is different between the two. A protocol implementation (e.g. a routing agent) usually has an *event loop*, which is an infinite loop that waits for data to arrive on one or multiple sockets, decodes the data to extract the PDUs, and processes the PDUs according to the protocol. As a result of the processing, new PDUs may need to be transmitted, at which point they are encoded as data and written to one or more sockets. Time based transitions are typically implemented using a system call to suspend the process for the required time, thus saving CPU cycles. For example, in UNIX systems it is frequent to see protocol implementations to use a *select* or *poll* based main loop, allowing them to wait for a certain elapsed time with the process suspended, but at the same time be notified when new data has arrived at a socket. In

a simulation environment, on the other hand, time is virtual, represented by a *virtual clock*, which is a simple numeric counter. Passing time in a simulator is represented by an event and does not require the process to be suspended during that time, only to increment the virtual clock by a certain amount. Events are also used to represent the reception of data from a network interface.

In the simulator, the *event scheduler* is essentially an infinite main loop that processes pending events in order. It is very similar to the *select*-based event loop in a protocol implementation. The differences are that 1) in the simulator, elapsed time is virtual, in the implementation it is real, and 2) in the simulator, a node simulates the reception of data from another node, while in the implementation the data is actually received from a real network interface. The protocol-specific aspects are common to the simulation and implementation, only the "environment interface" aspects are different. The question is whether we can adapt one to the other.

We propose Network Simulator 3 (ns-3) to be used as the basis for a new unified protocol development process that reuses the code of simulation for the implementation. Since version 3.2, ns-3 has received support for a "real-time simulator". The real-time simulator is an alternative event scheduler that can be selected at run-time. It synchronizes the virtual clock of the simulator with the actual real time of the host system where the simulation program is running. Thus, if an event is scheduled to happen in t seconds, then the callback function associated with the event will be called after exactly t seconds of real time have elapsed. Shortly after, in version 3.3, ns-3 received support for an "emulated NetDevice", or EmuNetDevice. In ns-3, a NetDevice is a class of objects that simulate an particular link layer type, such as Ethernet, WiFi, or point-to-point. The addition of EmuNetDevice has bestowed ns-3 with the ability to receive packets from a real network interface and convert them into simulated packets, as well as transmit simulated packets, generated by the simulation, through a real network interface. Emulation in ns-3 is not as difficult as, say, in ns-2. While in ns-2 packets are simulated as objects that do not know how to serialize themselves into a byte stream, in ns-3 packets are represented internally as bytes, like real packets, even in pure simulation mode, and the simulator uses *Header* classes in order to convert between PDU format and byte format. For this reason, emulation in ns-3 works with any protocol, not just a select few protocols prepared to support emulation, as in ns-2.

These are the main ingredients for enabling real protocol implementations to emerge from an ns-3 simulation model of that same protocol. We only need to build a simulation program, with one node only, and multiple

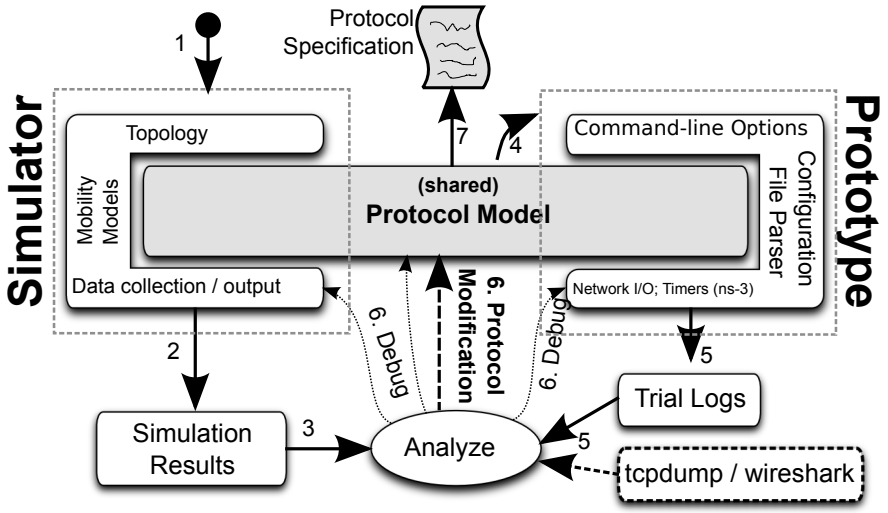


Figure 4.16: Proposed protocol development process

EmuNetDevice instances, one attached to each of the real network interfaces. The protocol simulation model runs on that single node, but remains unaware that it is running in emulation mode. Thanks to the ns-3 architecture and features, very little code is needed on top of the protocol simulation model to deploy it as a real-world implementation.

If the simulation and implementation of a protocol follow this approach, wherein most of the code is exactly the same for both, we may follow a different and improved development model, presented in Fig. 4.16:

1. From the scenario / problem description a protocol model is developed, along with a simulator that uses that model;
2. Multiple simulations are run, with varying parameters;
3. The simulation results are analyzed, the simulator is debugged, or the protocol (model) is improved;
4. The protocol model is reused to form the protocol prototype;
5. The protocol prototype is deployed in a testbed and trial logs are obtained. Packet traces (tcpdump / wireshark format) may also be collected;
6. The logs are analyzed and compared to the simulation results. As a result, the protocol may need to be improved or corrected;

7. Finally, when the protocol is working satisfactorily in both simulation and real world tests, the protocol specification document may be written.

The advantages of the single protocol module for both simulations and implementation are clear. First, development time is saved, since only a small (and very generic) wrapper needs to be written to run the protocol model as an implementation. Second, when the protocol is put to real-world trials and needs to be adjusted, the modifications are made into a single software module; this is simple and does not have the risk of introducing deviations between simulation and implementation models.

4.4.2 Ns-3 emulation

In ns-3, a *NetDevice* is the class of objects responsible for simulating a layer-2 network interface. In Linux systems, the EmuNetDevice subclass is available, allowing a ns-3 simulation to receive real packets from a real network interface, and to send simulated packets through the same network interface.

At the core of EmuNetDevice is a "packet socket" (PF_PACKET, SOCK_RAW) socket file descriptor. When the simulator asks the EmuNetDevice to send a packet (ns3 class *Packet*), the method `Packet::CopyData` is called, which extracts the packet contents into a byte buffer. Then, the sendto() system call is performed, using the packet socket file descriptor and the packet byte buffer as parameters. The code to receive packets, depicted with some simplifications in Fig. 4.17, is slightly more complicated. In fact, because we cannot block the main simulation event loop, nor is it efficient to make a poll/select system call between each simulation event iteration, a separate thread is created specifically to receive data from the packet socket. This thread runs an infinite loop that 1) allocates a memory buffer, 2) calls recvfrom() to receive the next packet, 3) schedules an EmuNetDevice method to be called from the main simulator thread, passing a pointer to the memory buffer as parameter. The method that is called in the main thread, by request from the receive thread, simply converts the raw memory buffer into an ns3 Packet (using an appropriate Packet constructor), releases the memory buffer, and informs the simulated node that it has received a new packet.

The conversion between ns3::Packet and raw memory byte array, and back, is trivial in ns-3 because ns-3 Packets always store simulated packets in a raw memory format. The only exception being that ns-3 Packets support

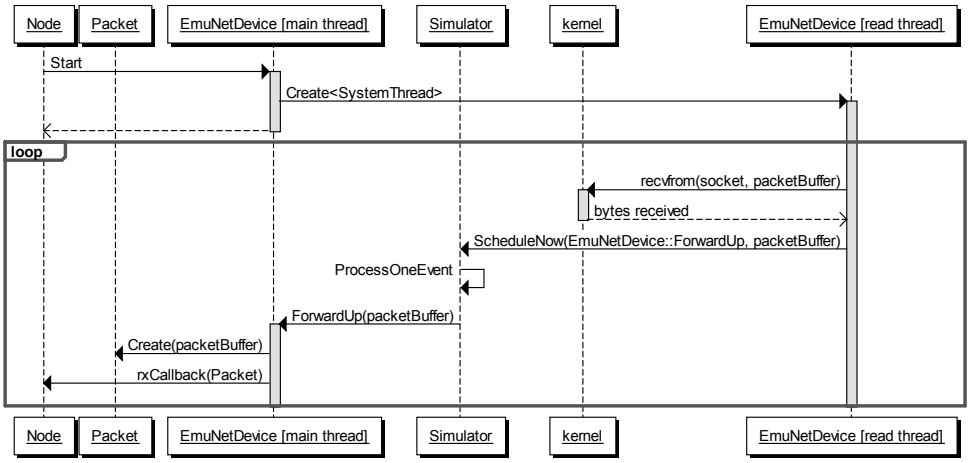


Figure 4.17: Ns-3's EmuNetDevice receiving packets from the network

a memory optimization wherein an application can choose to send "dummy" bytes, i.e. a block of bytes whose value is not important for the simulation. In this case, ns-3 avoids allocating memory for those bytes and just records the size and offset of that block of dummy bytes. Nonetheless, the conversion is much simpler and natural than in almost any other simulator.

With this emulation method, one can implement practically any kind of network operation in ns-3, as long as it works above layer 2. A question that remains, however, is how does this method fare performance-wise?

Performance evaluation

In order to better assess the computational and network performance footprint induced by using such emulation method in a real world scenario, a set of tests were defined. These tests focus aspects such as the impact on the host machine resources utilization and network performance/quality affectation, depending on the offered data rate and packet size used.

The hardware used to run the tests is a mini-itx Intel Atom D510. It has an x86 architecture for ease of use and better compatibility, avoiding the hassle of cross compiling. The hardware was chosen having in mind, also, a balance between cost, performance, power consumption and physical size, so it could be easily deployed in a real world scenario as a network element, performing operations such as routing and bridging. The operating system used was Ubuntu 10.04 x86.

The three scenarios implemented to perform the tests are presented in

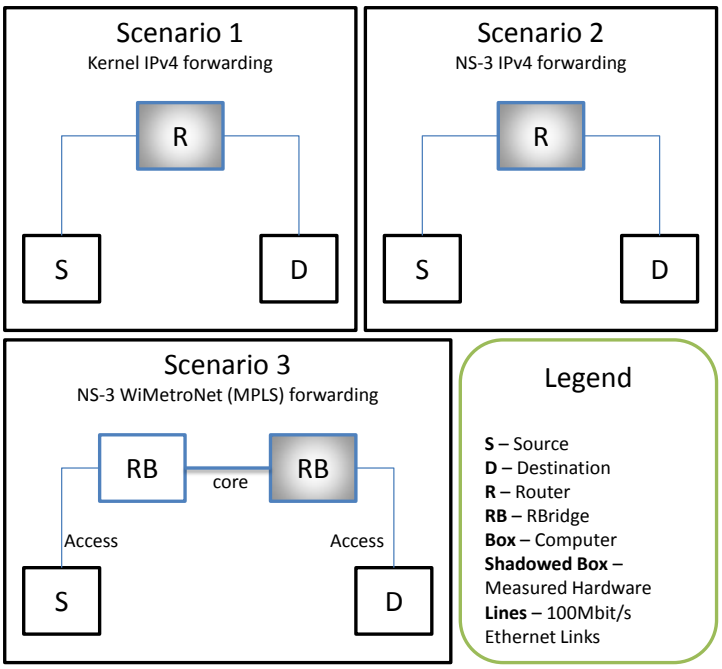


Figure 4.18: Data plane forwarding test scenarios

Fig. 4.18. While scenarios 1 and 2 were designed to compare IPv4 packet forwarding performance between kernel and ns-3 emulated implementation, scenario 3, on the other hand, is used to derive what performance would be expected from a custom implemented routing protocol, with its own data plane encapsulation operations.

Scenario 1 is composed of three nodes: 1) S runs an iperf client, generating an UDP flow to D. S also runs ping application, measuring the RTT while the flow is generated; 2) D runs an iperf server, receiving the UDP flow from S and calculating network statistics, such as received data rate and packet loss rate; 3) R represents the router, default gateway for S and D networks. It is the responsible to forward IPv4 packets between the two different networks. The IP forwarding operations are performed in kernelspace by enabling the IP forwarding option of the Linux kernel. Scenario 2 is similar to Scenario 1. The difference resides on node R. Now, the IP forwarding operations are performed by an ns-3 virtual node, in userspace, connected through EmuNetDevice's to the real network interfaces. Scenario 3 employs WiMetroNet RBridges elements, instead of an IP stack, which use the standard IPv4/Ethernet stack on the access networks and MPLS encapsulation in the core network. Two RBridge elements are used in order to introduce the need to perform "ingress" and "egress" packet operations, like it would be in a real world scenario.

For each scenario, we ran series of tests, gradually increasing the generated data rate between 1 and 90 Mbit/s[14], and with two different UDP payload sizes: 1400, and 160 bytes. Packets with 1400 bytes represent the usual application traffic over TCP. While the Ethernet MTU is 1500 bytes, given the UDP/IP payload, plus the encapsulation overhead in Scenario 3, an UDP payload size larger than 1400 bytes would risk fragmentation. The 160 bytes packets, on the other hand, are representative of typical VoIP traffic. Each test, lasting 30 seconds, was repeated 3–5 times for confidence interval purposes.

For each test, the received data-rate, average round-trip time, packet loss ratio, and CPU load were measured. The received data-rate and packet loss ratio were extracted from the iperf output statistical data. The average round-trip time was measured with the "ping" utility. The CPU load was measured at the shaded nodes in Fig. 4.18, using the "time" builtin command, computed as the sum of "user" and "system" time, divided by the real elapsed time. The CPU load can assume values in the range 0–2

[14]Approximately the maximum throughput attainable using IPv4 with UDP packets with payload size 1400 on 100Mbit/s Ethernet links

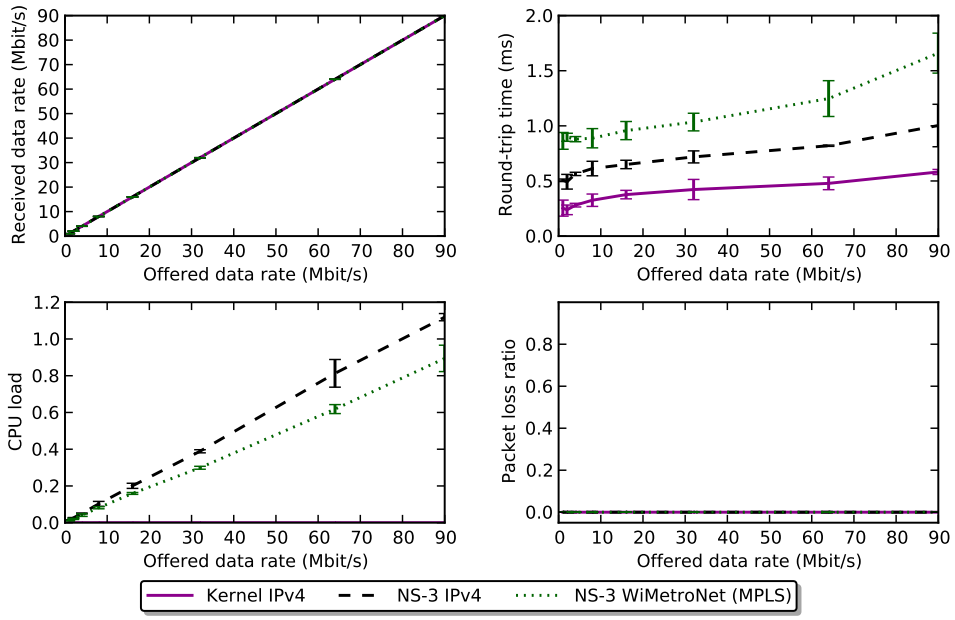


Figure 4.19: Dataplane forwarding results for 1400-byte packets

because the node has a dual-core processor and ns-3 has a multi-threaded design for implementing Emulated Network Devices. For Scenario 1, the CPU load was not measured because it revealed to be negligible, even when performing full link speed forwarding with small packets.

The obtained results are represented in Fig. 4.19 and Fig. 4.20, for packets of 1400 and 160 bytes, respectively. For 1400-byte packets, both scenarios were able to attain a data-rate of 90Mbit/s without any packet loss. While in kernelspace the CPU load was insignificant, in userspace the CPU load increases linearly with the offered data rate, reaching approximately 1.1. The round-trip time (RTT) measured in the scenarios running ns-3 increased roughly 0.4 ms, due to the user space processing of each packet. For a 1400-byte packets, despite the high CPU load, the obtained results can be considered very good, since the additional delay introduced has remained below 1 ms. For an MTU of 160, the plots in Fig. 4.20 show clearly a very high performance difference between Kernel and ns-3 IP forwarding solutions. In the Kernel scenario it was possible to reach a data-rate of, approximately, 60Mbit/s, which is the maximum throughput of 100Mbit/s Ethernet link for an MTU of 160. The RTT remained very low and stable and the CPU load was also insignificant. Although the offered data-rate

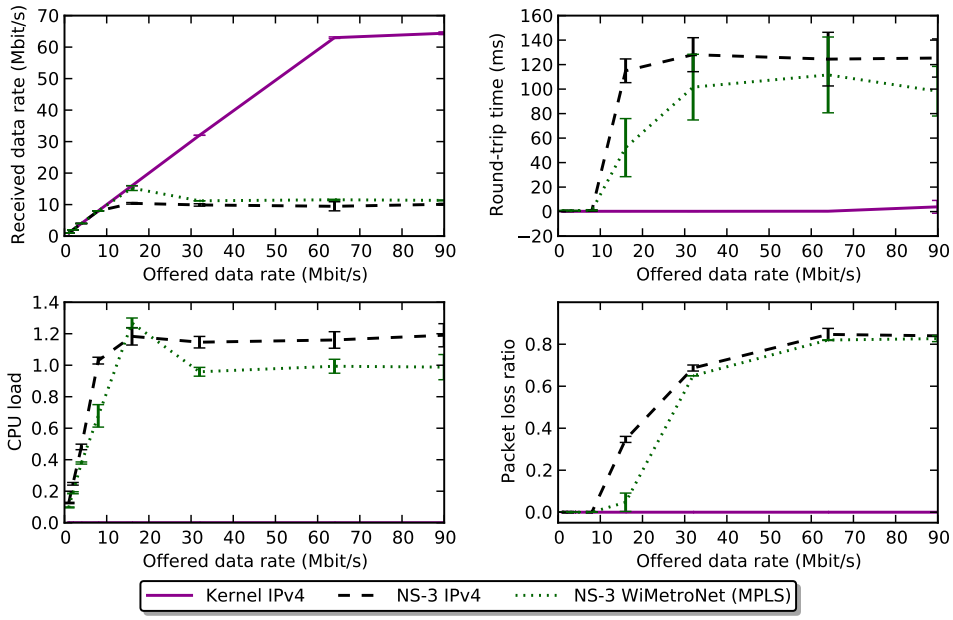


Figure 4.20: Dataplane forwarding results for 160-byte packets

was configured up to 90Mbit/s in iperf, this did not impact the packet loss results in this scenario, because iperf client is aware of the link capacity and does not attempt to transmit more than the link capacity.

In contrast, Scenario 2 reached only a data-rate of around 10Mbit/s, which is approximately six times slower than the kernel implementation. This happens due to the high packet number that has to be processed in userspace. It becomes evident, now, that in these kinds of emulated scenarios the bottleneck is defined by the number of packets to be processed, and not so much by their size. The RTT value stabilized at around 120ms, while the packet loss was always increasing because of a limit in ns-3 for the number of packets waiting in memory to be processed. While, at first sight, these results could seem unsatisfactory, if we consider that these 10Mbit/s are representative of VoIP traffic, which consume very small bandwidth (less than 64 kbit/s) per-flow, it actually represents a very large number of VoIP flows.

Finally, the results of the third scenario shown that it performs better than the second one, attaining lower CPU usage per packet processed which resulted in better measured data-rates and RTTs. For the 1400-byte packet plots in Fig. 4.19, the average RTT of Scenario 3 appears to be worse than

the Scenario 2, but it is necessary to point out that each packet was then subjected to two forwarding operations in each direction, instead of only one. The better results obtained by this scenario could be mainly due to the fact that RBridges forwarding operations take place at layer 2.5, not using the ns-3 layer 3 implementation and its associated computational overhead.

The results confirm the hypothesis that the kernel to user-space context switch and data transfer are the main bottleneck. In other words, most of the performance issues associated with ns-3 are related to the number of packets to handle, and not so much with the size of those packets. Thus, the proposed implementation method can handle reasonably well application flows dominated by large packets, such as any TCP based protocol (HTTP, FTP), but does not handle so well high bitrates and small packets. Fortunately, the combination of high bitrate and small packet size is not very common. It is true that VoIP (voice over IP) flows are composed mainly of small packets, but each of those flows consumes a small bitrate. Only a high number of simultaneous VoIP flows will be able to saturate a network link with a few tens of Mbit/s. For this type of scenario, implementing a data plane using ns-3 is not recommended.

4.4.3 Ns-3 control-plane emulation

It is a well known fact that the implementation of a data plane in userspace is not generally recommended, at least for a final implementation of a protocol, due to the performance degradation it entails, as shown in the previous section. On the other hand, generally there is not significant performance degradation with implementing control plane functions in a userspace daemon; it is actually common practice, and recommended from a security standpoint. With respect to performance, userspace implementations are adequate due to the fact that the rate of packets to be processed in the control plane is generally low, no more than 5–10% of the link bandwidth. Additionally, most well designed protocols have a mechanism to aggregate multiple messages in a single packet, thereby reducing the number of packets to process. As an example, if we consider a grid of OLSR routers, with a layer-2 MTU of 1500 bytes and a total network size of 500 nodes, each node receives 5 packets per second for the case of 4 neighbors per node, and 9 packets per second if the number of neighbors is 10. Any userspace daemon can handle this packet rate with ease.

UDP control plane optimization

When implementing a control plane protocol in ns-3, EmuNetDevice actually captures all traffic that is received by a network interface: not only our protocol packets, but also data plane and traffic of other control plane protocols exchanged in the network. All the packets we are not interested in are eventually dropped by ns-3. However, before being dropped they consume considerable processing time: the context switch to let the userspace program receive the packet, and some processing by ns-3 (some headers decoded). This overhead could be avoided if ns-3 could use real sockets that would receive only the specific protocol number we are interested in. Since most new research protocols are carried on UDP payload of a registered port number, we have developed a real (system) udp socket module for ns-3.

In our implementation[15], we have developed the classes *RealUdpSocket* and the respective factory, *RealUdpSocketFactory*. RealUdpSocket allows ns-3 applications to talk to real UDP sockets using the usual ns-3 Socket API with no modifications whatsoever. As basis for the implementation, a strategy similar to EmuNetDevice is used, i.e. the main simulation thread is used to send data, while a separate thread waits for incoming data. In addition, we created a *RealStackHelper*; it works similarly to InternetStackHelper, but instead of UdpSocketFactory it adds a RealUdpSocketFactory object to a Node, and additionally adds "dummy" interfaces to the Node, mirroring the real host interfaces, MAC addresses and main IPv4 addresses included. The dummy interfaces with real MAC and IPv4 addresses are useful to allow ns-3 routing protocol models to run unmodified even when those protocols need to discover the list of interfaces and IPv4 addresses that exist in the Node where they are running. These classes are illustrated in the class diagram in Fig. 4.21.

Thanks to the *RealStackHelper* class, adapting a routing protocol to run in emulation mode using the more efficient *RealUdpSocket* becomes extremely simple, as shown in the program listing below. After replacing *InternetStackHelper* with *RealStackHelper* (see lines 16–18), the simulation script no longer needs to configure network interfaces in the node, as they are automatically configured based on the real network interfaces in the system. Without RealStackHelpr, the three lines 16–18 would have to be replaced by approximately 36 lines of code that would read from the command line arguments the IPv4 address, and mask of the host interfaces that we wish to configure, and create an EmuNetDevice instance for each. Moreover, passing these parameters through the command-line is both tedious and error

[15]It is available online at the URL: http://code.nsnam.org/gjc/ns-3-real/

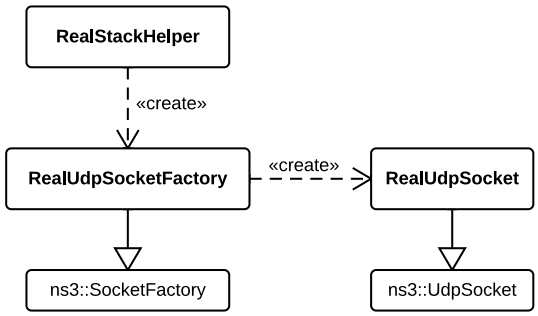


Figure 4.21: Overview of additional ns-3 emulation classes developed

prone.

```
1  int main (int argc, char *argv[])
2  {
3    GlobalValue::Bind ("SimulatorImplementationType",
4        StringValue ("ns3::RealtimeSimulatorImpl"));
5    GlobalValue::Bind ("ChecksumEnabled", BooleanValue (true));
6
7    Ptr<Node> realNode = CreateObject<Node> ();
8
9    OlsrHelper olsr;
10   Ipv4StaticRoutingHelper staticRouting;
11
12   Ipv4ListRoutingHelper list;
13   list.Add (staticRouting, 0);
14   list.Add (olsr, 10);
15
16   RealStackHelper stack;
17   stack.SetRoutingHelper (list);
18   stack.Install (realNode);
19
20   Simulator::Run ();
21
22   return 0;
23 }
```

Performance evaluation

In order to evaluate the performance of control plane implementation using ns-3, we used the open source olsrd daemon (from olsr.org), and wrote an ns-3 based OLSR agent.

The olsrd implementation[16] is a mature open source implementation of the OLSR protocol, under development since at least 2004. We modified it in two ways: 1) added some minimal instrumentation code to count the

[16]Snapshot taken from the stable Git branch on 18$^{\text{th}}$ May 2010.

number of HELLO messages processed, and print that number at the end, 2) removed the "CPU overload" limit of maximum 32 messages processed between loop iterations (function `olsr_input()` in `src/parser.c`); the reason for the limit is not documented. In any case, it was severely limiting the performance of olsrd, so we had to remove it to obtain more accurate readings.

The ns-3 based OLSR implementation (ns-3-olsrd) uses the unmodified builtin ns-3 OLSR model (`src/routing/olsr`). This OLSR model requires only: 1) ns-3 UDP sockets, 2) access to an Ipv4 object on the node, to query the list of interfaces and respective addresses, and 3) an event scheduler, for timers. The ns-3-olsrd program actually has a command-line switch that selects between two approaches of implementation: EmuNetDevice and RealUdpSocket. The version that uses EmuNetDevice accepts a list of (interface name, IPv4 address, IPv4 network mask) and, for each interface it creates, an EmuNetDevice attached to that interface, while registering the respective address/mask for that interface. The RealUdpSocket variant just uses RealStackHelper, which automatically scans the network interfaces and adds the real udp socket factory to the node. The entire ns-3-olsrd.cc source code, containing the two implementation approaches, is actually only 122 lines long.

Speed Our test scenario was composed of one "test" node running OLSR, and a variable number of neighbors in the same LAN, sending HELLOs at the normal rate (one every 2 seconds). These neighbor nodes are simulated by a single Linux host which runs a HELLO generator, with the IP address of each HELLO varying to simulate the multiple neighbors. Since the neighbors only generate HELLOs, but do not listen to HELLOs from the other (test) node, the test OLSR node will see the neighbors as having an asymmetric link to it. In this way, the test OLSR node does not have to recompute any routing table with each HELLO received; it only needs to receive the packet, decode it, and record the neighbor IP address and interface (a "link tuple") in an internal data structure (the "link set"). Thus, we are only measuring the speed of the OLSR implementation at receiving a packet, decoding the headers, and minimal processing. This is exactly what we want to measure. The time taken to compute the routing table after a topology change is not related to the protocol development framework and therefore irrelevant in this context.

We ran a series of tests, gradually increasing the number of OLSR neighbors, and measured the number of HELLOs that a particular OLSR imple-

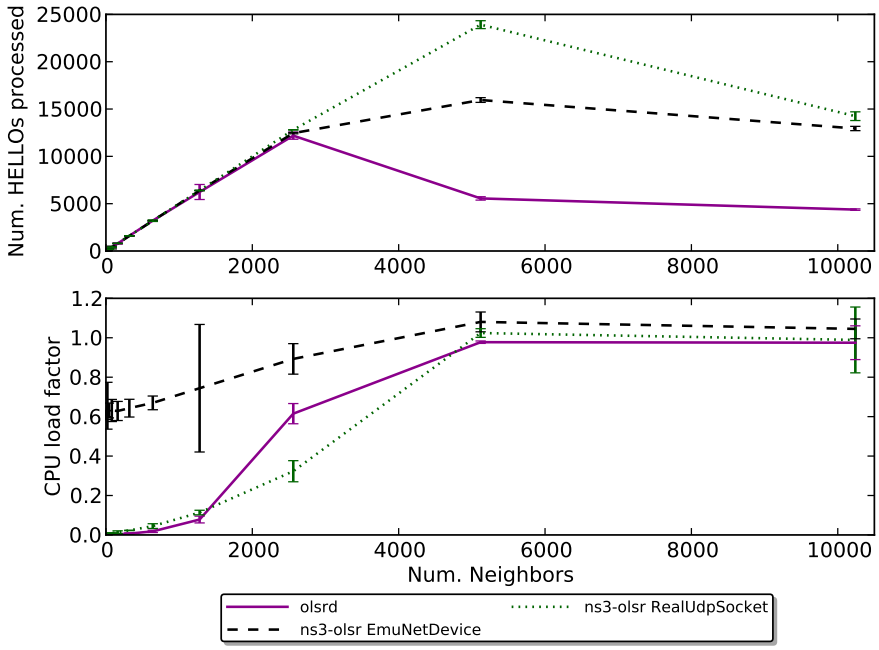


Figure 4.22: Control plane (OLSR) performance test results

mentation was able to process in a period of 10 seconds. For each of the the OLSR implementations — olsrd, ns-3 with EmuNetDevice, and ns-3 with RealUdpSocket — and for each topology (number of neighbors), the experiment was repeated three times for confidence interval purposes. We also measured the CPU load (fraction of CPU utilization) of the OLSR implementation. While the tests were running, we had background TCP traffic (iperf) to saturate the network link.

From the results in Fig. 4.22 we can see that the OLSR implementations process a number of messages proportional to the number of neighbors, but they all eventually reach a point of overload, after which they can no longer process more messages. We do not have enough data samples to know the exact limits, but for olsrd the limit is between 2560 neighbors and 5120 neighbors. The peak packet processing rate was 1280 packet/s for 2560 neighbors, but then it diminished considerably, achieving only 437 packet/s with 10240 neighbors. The scalability issues are probably due to less efficient data structures used by olsrd to store the link tuples. The ns-3 EmuNetDevice implementation also started to lose HELLOs between 2560 and 5120 neighbors, but achieved a maximum processing rate of about

1600 packet/s with 5120 neighbors, decreasing to 1294 packet/s with 10240 neighbors. Finally, ns-3 with RealUdpSocket starts to lose packets only between 5120 (2392 packet/s) and 10240 neighbors (1424 packet/s).

Interesting also are the CPU load results, where we can see that the maximum HELLO processing rate is directly linked to the point at which each implementation starts consuming 100% of the CPU time. The ns-3 EmuNetDevice implementation is the one that consumes more CPU time, which can be explained by the fact that it is processing not only HELLO packets but also the background traffic. In fact, for high load it even surpasses 100% CPU time because it uses two threads and the system has a dual-core CPU. The ns-3 RealUdpSocket implementation tends to be slightly more efficient than olsrd, and considerably more efficient than ns-3 with EmuNetDevice.

Size/memory Besides the processing speed, another performance dimension that is interesting is regarding memory requirements. Comparing the program executable binary files we found that olsrd has a binary file size of 296 KB, while ns3-olsrd, when compiled statically, had a file size of 13 MB. One of the reasons for the large file size of the ns3-olsrd version is that ns-3 contains many simulation models, the OLSR routing protocol being just one among many. To reduce the file size, we tweaked the ns-3 build scripts to disable most of the unneeded ns-3 modules, and this way the program file was reduced to only 4.4 MB[17].

In order to find out actual runtime memory requirements of the implementations, we repeated the HELLO processing tests while measuring the maximum resident memory size[18] of the processes. The results, in Fig. 4.23, show that the ns-3 version consumes about 13-14 MB of memory right from the start, but the memory consumption increases very slowly. In contrast, olsrd starts by consuming only 4.2 MB, but the memory increases quickly for increasing number of neighbors. Clearly, olsrd would eventually surpass ns3-olsrd in memory consumption if the initial growth trend was maintained. But, as we have seen, olsrd is not able to process all the HELLOs between 2560 neighbors and 5120 neighbors, and for this reason the memory consumption stops growing. At maximum load, ns-3-olsrd still consumes around 14 MB of memory, while olsrd takes around 12 MB.

We do not know with certainty what contributes to ns-3 consuming more

[17]Removing the builtin ns-3 unit tests, the binary was further reduced to 3.5 MB, but we did not develop this optimization until after all the tests were done.

[18]The maximum resident memory size (RSS) is the maximum amount of memory actually allocated by the kernel for the process as physical memory, and it was measured using the Ubuntu program `/usr/bin/time`.

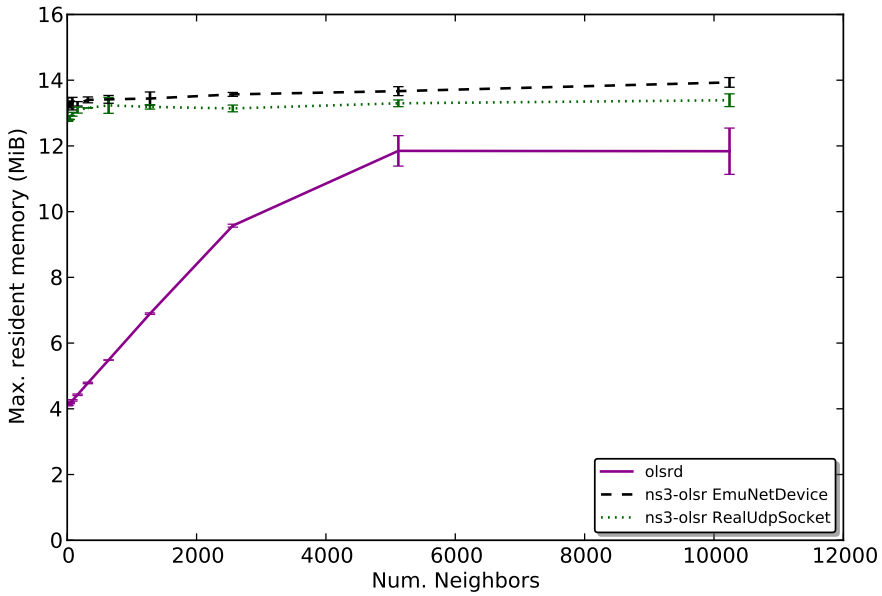


Figure 4.23: Control (OLSR) memory test results

memory, but one possible explanation is that the ns-3 runtime, even when stripped to the fundamental modules, contains a lot of useful code that is made available to the protocol developers, even if not used. Second, the C++ language itself tends to produce larger compiled programs than pure C programs, especially when template programming is used extensively, as is the case in ns-3, because the compiler has to instantiate the generic code for each type used. This leads to more code generated, although it tends to run faster, since it is specialized for each type. Third, the ns-3 EmuNetDevice and RealUdpSocket implementations create a separate thread to handle reads. Multiple threads share the same data segment, but each thread requires a private stack segment. In recent Linux systems, the maximum stack size is 8 MB by default, and programs typically take a few hundred kB of stack, per thread. Finally, the memory we measured includes both private and shared resident memory. The standard C++ library (libstdc++) accounts for 920 kB of virtual memory, 432 kB of which are resident. Olsrd does not use this library and so does not incur its overhead.

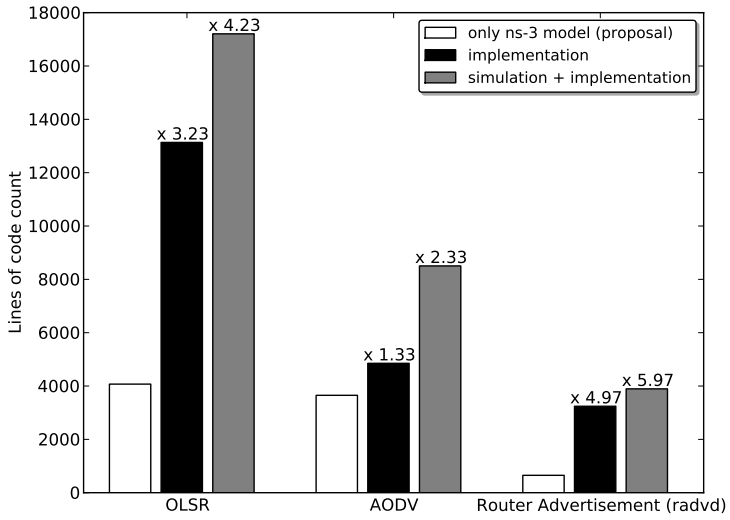


Figure 4.24: Writing protocols for ns-3 alone is faster

Development effort reduction

We have evaluated the performance of the ns-3 method of developing protocols, when compared to a dedicated implementation and concluded that ns-3 is adequate to write userspace protocols. But the main point of the proposed approach is to reduce the development effort of new protocols. In an attempt to measure this reduction, we measured the number of lines of code of three protocols, in their ns-3 and real-world implementation variants. We used the "cloc" open source tool to count only lines of code, excluding blank lines and comments. The results in Fig. 4.24 include three bars for each protocol. The first bar represents the size of our proposed ns-3 model, which can be used for either simulation or, with just an additional *main* wrapper with about 100 lines, deployment in a real network node. The second bar represents the line count of a corresponding real implementation. When developing a new protocol from scratch, an implementation is not enough to develop a protocol; first we need to write a simulation model in order to run simulations. Thus, a third bar is included that represents the sum of a simulation model (in this case from ns-3) and real implementation line counts. For measuring the ns-3 models we used ns version 3.9. The OLSR protocol real implementation again comes from olsrd (same version as before). The AODV implementation used was `aodv-uu-0.9.6`, and the radvd used was the `radvd-1.6` source from Ubuntu 10.10.

It is interesting to find out what most contributes to the line count

difference between the two versions. In the case of OLSRd, to make the comparison fair we only included the most generic OLSR protocol code, excluding OS-specific functions and functionality that are not present in the ns-3 OLSR model, such as IPv6 address support, plugin system, advanced "link quality" metrics (such as ETX [87]), "Smart gateways", ipv4 over ipv6 traffic, and memory allocation statistics. But even without counting those subsystems, OLSRd has about 2.13 times the lines of code of ns-3's OLSR model. One reason for this difference is that OLSRd is a standalone program that does not use any external library, and so has to write from scratch many functions that are already included in ns-3, such as sequence numbers (that can "wrap around" and still be compared), IPv4 addresses, packet buffers and writing to packets with iterators. This problem could be solved by writing a "networking library" that OLSRd could use[19]. Similarly, being written in C, the OLSRd program needs more lines of code to deal with containers (linked lists, balanced trees, hash tables), which are taken for granted in C++ with STL containers, in addition to simpler memory management of the ns-3 Object system and smart pointers. Finally, we are forced to acknowledge that the real OLSRd implementation is more detailed, with features to improve scalability (e.g. using binary tree data structure for duplicate detection) and security (e.g. checking for unusually large increments in message sequence numbers). This of course consumes more lines of code, but they are not easy to discard for comparison purposes.

Although line counts do not always translate linearly to development effort, from these results we get a rough estimate that writing protocols for ns-3 alone is substantially faster than writing first a simulation model and then a dedicated implementation. Not only is writing in ns-3 usually simpler than a dedicated implementation, since we get to reuse the ns-3 protocol development framework instead of writing one from scratch, but also avoids the need to program the same protocol twice.

4.4.4 Related work

The idea of using a protocol simulation model as implementation is not new. However, there are issues with the previous attempts.

The highly popular ns-2 simulator does have some support for emulation, but only some protocols are prepared for this emulation mode. Moreover, ns-2 does not use real IP or MAC addresses, making the emulation more complicated. Finally, ns-2 does not have a software architecture as clean as

[19] Although we may consider ns-3 itself as such a "networking library".

other existing simulators, and does not provide a very healthy environment to develop new protocols, for reasons previously stated.

The Click Modular Router [88] is a graph-oriented architecture for building a router by assembling a graph of elements. Each element has a well defined set of input and output ports for exchanging data with other elements, and performs a well defined and small function, such as decrementing the TTL of a packet. Extensions for running Click inside the ns-2 [89] and ns-3 simulators exist, allowing for a hybrid implementation/simulation environment. However, in Click the protocol to be developed must be written as a set of Click elements and a configuration file provided for creating a graph that connects those elements. This forces the researcher to learn two different programming environments and a new configuration file syntax. Moreover, Click uses *flow-based programming* [90], which is a unusual programming paradigm for researchers that only had prior experience with discrete event simulators and has, consequently, a learning curve. We may also observe that existing Click based protocol models employ a very high number of classes, each class implementing one element, with very little functionality, leading to a lot of "boilerplate" code. Another problem is that Click uses pointers, instead of smart pointers, making memory leaks very easy to be introduce by non-expert programmers.

RapiNet [91] is defined as a "development toolkit for rapid simulation, implementation and experimentation of network protocols". It uses a *declarative networking* approach for defining new protocols, employing a language called *Network Datalog* (NDlog), which extends *Datalog*, a query / rule language. The RapidNet compiler is able to generate ns-3 code for either simulation or emulation mode, thereby achieving the goal of using the same model source for both simulation and implementation. The NDlog programming language, however, is not Turing-complete, meaning that it may not be able to compute everything that Turing-complete languages, such as C++, can compute. Moreover, the language uses a unusual paradigm to describe protocols that has a steep learning curve.

Another interesting tool for integrated simulation/implementation is the *Protean Protocol Prototyping Library* (ProtoLib), developed at the Naval Research Laboratory. It allows the same code to be built for a simulator (ns-2 and OPNET) and run as a standalone implementation. On the other hand, to do this the protocol needs to be written using the ProtoLib API. Besides having to learn a new API (which does not even use smart pointers), researchers will then be unable to submit the new protocol model for inclusion into existing simulators.

The OPNET Modeler is a simulator that also allows integration of pro-

	Proto. compat.	**Devel. difficulty**	**Performance**
ns-2 emulation	Partial	Intermediate	Average
Click	Full	Intermediate	Good
RapidNet	Partial	Intermediate	Average
ProtoLib	Partial	Intermediate	Average
OPNET Modeler	Partial	Intermediate	Average
ENTRAPID	Full	High	Good
ALPINE	Full	High	Good
NSC	Full	High	Good
IMUNES	Full	High	Good
ns-3 emulation	Full	Low	Average

Table 4.5: Comparison of unified simulation/implementation approaches

tocol models with real systems, thanks to its "System-in-the-loop" functionality [92]. However, its closed and commercial nature invalidates it being a viable alternative in many contexts. For instance, not having the full source code available precludes porting it to certain router architectures.

Some tools work with some form or subset of an operative system kernel code. The ENTRAPID [93] project virtualizes just the networking portion of a BSD kernel, thereby enabling hundreds of kernel instances to run on the same system, connected by virtual links. A similar approach is followed by ALPINE [94]. The Network Simulation Cradle (NSC) [95] also virtualizes a kernel, but instead of allowing real applications to communicate over the network of virtualized kernels, it embeds the virtualized kernel instances into simulated nodes of an existing simulator (ns-2 and ns-3 supported), and supports multiple kernel stacks, not just BSD. IMUNES [96] also virtualizes the kernel networking code, but creates virtual nodes and virtual links inside the kernel, instead of userspace, to avoid frequent context switching and achieve greater efficiency.

All these kernel based approaches have the same basic problem: they require new networking protocols to be developed inside the source code and framework of one of those operative system kernels. That code base, although highly detailed, realistic, and well optimized, is not a very programmer-friendly environment. Moreover, protocols (at least control plane protocols) are almost always developed to run as userspace daemons anyway, for security reasons, so the effort to develop using a kernelspace API may not be worth it.

Table 4.5 evaluates the each protocol development framework acording

to three main properties:

1. Protocol compatibility: this property indicates if the complete range of protocol models already written in the underlying framework is supported. In this regard, ns-2, RapidNet, ProtoLib, and OPNET Modeler require protocols to be written differently from the underlying simulator if they are expected to be deployed in a testbed in emulation mode, while in the other frameworks the protocol model is written in the same style as any other protocol. Full protocol compatibility makes it easier to include a new protocol into the main simulator release, for other researchers to use, while not sacrificing the ability to deploy in real nodes;

2. Development difficulty: this property reflects the obstacles that are faced by a programmer to develop a working protocol in the framework, such as memory management facilities, API readability, object-orientation, and simply number of lines of code required to implement a protocol. In this regard we find ns-2, Click, RapidNet, ProtoLib, and OPNET Modeler present considerable development difficulty, either due to low level and manual memory management required, unusual programming language/paradigm, or high number of classes that the programmer has to write. The kernel-based solutions, ENTRAPID, ALPINE, NSC, and IMUNES, have increased difficulty due to the requirement to write in a kernel framework[20]. Finally, in the case of ns-3, development is made much easier than in other frameworks;

3. Performance: this property roughly evaluates the real-world (not simulated) performance of a protocol implemented in each framework. We may find that all userspace frameworks, ns-2, RapidNet, ProtoLib, OPNet Modeler, and ns-3, offer roughly the same average performance. There are probably some performance differences between them, but quantifying these differences would require extensive benchmarking. Frameworks that run in kernelspace are naturally expected to have higher performance. This includes Click, which can run in either userspace or kernelspace.

[20]For practical and performance reasons, an O.S. kernel is usually not cleanly designed and well structured.

4.4.5 Summary

We addressed the topic of network protocol development methodology. The traditional protocol development process is reviewed, and the main problems associated with it are brought to light. One recurring problem is the duplication of effort to write first simulation and then implementation code. Another potential problem is the propensity for behavior differences being accidentally introduced between the two versions, leading to simulations offering results different from the implementation. We offer an alternative development process that takes advantage of the builtin network emulation features of ns-3. The ns-3 based development process allows developers to write a single model for the protocol that can be both simulated and deployed in a real node. The main difference between ns-3 and other similar frameworks is that in ns-3 everything can be done in C++ and using good C++ programming practices for ease of development.

In order to support the proposal to use ns-3 for implementation of new protocols, the performance of ns-3 running in emulation mode has been evaluated. The results show that the ns-3 IPv4 stack, in emulation mode, is able to process packets at a rate high enough to exhaust an 100 Mbit/s Ethernet link, when handling large packets, but can have problems forwarding traffic if it is composed mostly of very small packets. Nonetheless, we reckon ns-3 is still useful for small packet networks provided that the data plane is implemented in kernelspace and ns-3 handles only the control plane. This scenario was tested by comparing the ns-3 OLSR model with the open source OLSRd implementation, and we found that ns-3 can even outperform OLSRd under high load, albeit with a little more memory consumed.

We additionally contribute a new UDP socket emulation class — RealUdpSocket — that improves performance by allowing ns-3 to avoid processing background traffic packets, processing only the control plane packets. While theoretically RealUdpSocket could also be used for the full data and control-plane stack, the performance in this scenario would tend to become similar to what is obtained using EmuNetDevice, since it would then be processing all the packets, not just the control plane packets. To make deploying protocols simpler and more robust, the new RealUdpStack class may be used, as it automatically scans the real host's network interfaces, and registers with ns-3 equivalent "dummy" interfaces, with matching MAC and IPv4 addresses. This way, if an ns-3 protocol model is using the ns-3 UDP sockets API correctly, it can be easily deployed without further modifications.

4.5 Conclusions

In this chapter, we have addressed the ways in which the ns-3 simulator can be used to speed up research and development of new network protocols. First we have shown that a new protocol development process that takes advantage of the builtin network emulation features of ns-3 allows developers to write a single model for the protocol that can be both simulated and deployed in a real node. The performance of ns-3 running in emulation mode has been evaluated, and shown to be able to process packets at a rate high enough to exhaust an 100 Mbit/s Ethernet link, when handling large packets, although it can have problems forwarding traffic if it is composed mostly of very small packets. We also concluded that for implementing only the control plane part of a network stack, ns-3 is more than adequate, performance-wise, especially with our new contributed UDP socket emulation class, that improves performance by allowing ns-3 to avoid processing background traffic packets, processing only the control plane packets.

We have also made a set of additional contributions to ns-3 in order to make development of simulations easier and quicker. One such contribution is the "flow monitor" ns-3 module. This module allows developers to measure common flow metrics with only a few lines of code and with great runtime efficiency. Another contribution has been the Python bindings, allowing developers to write simulation scripts much faster than what the C++ programming language allows, while at the same time forging a new "PyBindGen" tool, independent of ns-3, to allow creating Python bindings for C or C++ code that are more efficient, portable, and easier to write than with other similar tools. Finally, we developed a tool to visualize and interact with a running simulation graphically, allowing the researcher to quickly find some simple mistakes, thereby saving significant development time, while giving more confidence that the simulation is behaving as intended. The visualization tool is shown to have very unique features, not found in any other visualization tool, derived from its "live" visualization approach.

Chapter 5

Conclusions

5.1 Work review

In the context of this thesis, we studied some technologies that are relevant for developing a network for metropolitan public transport systems where end user mobile terminals are strictly legacy terminals supporting WiFi and IPv4 with DHCP. We explored relevant link layer technologies, including WiFi (including the 802.11p variant), Ethernet, WiMax, and UMTS, to conclude that each has different strengths and weaknesses and it is beneficial to design a system that can take advantage of several link layer types. At the networking layer, we have studied 802.1D (Learning Bridge), the basic IP networking, the OSPF link state routing protocol, the OLSR and AODV adhoc routing protocols, MPLS, TRILL, and 802.11s. The 802.1D bridges use a very simple forwarding algorithm as long as the network is restricted to a tree topology, but does not scale well because of STP and broadcast storms. The Internet Protocol, IP, scales very well, but requires careful manual planning. OLSR and AODV enhance IP to avoid the need for network planning, but do not scale to networks of hundreds of nodes. The MPLS approach is very interesting as data plane, as it is both fast and flexible, but it needs a good control plane that can handle dynamic network topology changes. The TRILL concept of "virtual LAN" and encapsulation approach is interesting, but it is not designed to handle mobility of nodes and does not solve the broadcast storm problem effectively. We also studied several existing solutions, at different layers, for supporting mobility of end user terminals. SIP solves the mobility problem at application layer by having the terminal send a new SIP INVITE message to the CN right after handover. HIP works at "L3.5"; it defines a "host identity" layer, and

allows end hosts to obtain cryptographic host identifiers, independent of the current IP address of the node, allowing the IP address to change while the host identity to remains the same. The Mobile IP extension to IP works directly at layer 3 and requires no modification in applications, only in the IP layer. It allows nodes to have two addresses, one "home" address, denoting the identity of the node, and one "care-of" address, indicating the current location of the node. The mapping between home and care-of addresses is kept in a Home Agent node located in the terminal's home network. Fast Handover further optimizes MIP by allowing the terminal to prepare handover earlier, and tunneling packets from the old AR to the new AR for some time. The Proxy MIP approach is also interesting because it implements all the mobility signaling and tunneling work entirely within the network side, lifting the burden of implementing mobility from the terminal, and therefore better supporting legacy terminals. Mobility solutions implemented at the link layer level, such as in UMTS networks, also allow the terminal to switch Point of Attachment (PoA) transparently to the IP layer, i.e. the IP address of terminal never changes. Finally, we examined tools to enable rapid protocol simulation/deployment cycles. To that end, the design and limitations of common simulators has been described, including ns-2, ns-3, OPNET Modeler, OMNET++, PARSEC, and JiST/SWANS. Ns-3 is a rewrite of ns-2 from scratch, trying to solve many of the problems in ns-2. It has an open-source friendly license (GPL), is very efficient, and has a special focus on realism. Moreover, it has strong packet-level emulation abilities and real-time scheduling option, which can be exploited to make real-world experiments using the same protocol model developed for simulation.

A new proposed architecture for the public transport vehicle networking scenario introduced in Chapter 1 was presented. The new architecture was named "WiMetroNet". This architecture comprises a data plane and routing protocol designed to scale for large networks. It filters broadcasts and optimizes DHCP and ARP traffic via close integration of those protocols with the routing protocol, and encapsulates user frames using an MPLS header. The new routing protocol borrows design from OLSR, supports both mobile Rbridges and mobile end-user terminals, and feeds the data plane with IP/-MAC association tables, much needed for the DHCP/ARP optimizations. We then devised routing optimizations to handle fast-handover of terminals in an efficient and scalable way, for large networks. The new proposals have been demonstrated via simulation and analytical models, and the limits of scalability assessed for two different scenarios: a "road scenario", and a "city grid".

Taking advantage of the experience garnered during WiMetroNet re-

search and development, we then addressed the topic of network protocol development methodology. The traditional protocol development process was reviewed, and the main problems associated with it, brought to light. We explored an alternative development process that takes advantage of the builtin network emulation features of ns-3. The ns-3 based development process allows developers to write a single model for the protocol that can be both simulated and deployed in a real node. To support this proposal, the performance of ns-3 running in emulation mode has been evaluated. The results show that the ns-3 IPv4 stack, in emulation mode, is able to process packets at a rate high enough to exhaust an 100 Mbit/s Ethernet link, when handling large packets, but can have problems forwarding traffic if it is composed mostly of very small packets. We additionally contribute a new UDP socket emulation class, *RealUdpSocket*, that improves performance by allowing ns-3 to avoid processing background traffic packets, processing only the control plane packets. To make deploying protocols simpler and more robust, the new RealUdpStack class may be used, as it automatically scans the real host's network interfaces, and registers with ns-3 equivalent "dummy" interfaces, with matching MAC and IPv4 addresses. We made additional contributions to ns-3, all sharing the goal of making protocol development faster and easier. The ns-3 Flow Monitor framework allows developers to measure the most important metrics of data flows in simulations with just a few lines of code, saving a lot development time, but with small runtime overhead. The ns-3 scripting framework, in the form of ns-3 Python bindings, enables researchers to write simulation scripts faster. To support the ns-3 Python bindings, a new tool called PyBindGen was developed to enable developers to bind C++ libraries to Python. The ns-3 Python bindings are easy to maintain due to the automatic C++ header scanning ability of PyBindGen, cover most of the ns-3 C++ API, and do not deviate from the C++ API, making switching between the two languages relatively straightforward. The PyBindGen tool was evaluated in terms of performance and was found to have much better performance than Boost.Python or SWIG. PyBindGen has similar performance to SIP, but is much more easily extensible, via Python plugin code, and is more portable. Finally, we developed a new visualization tool for ns-3, to make debugging of protocol models and simulation scenarios easier.

5.2 Original contributions

The following contributions have been produced during the course of this work:

1. WiMetroNet: a novel network architecture designed for a public transportation system that is scalable and appears to end user terminals as just a large Wireless LAN segment. This includes:

   (a) A new scalable data plane that (i) solves broadcast problems by forbidding broadcasts in general and providing only DHCP and ARP support in a way that does not require flooding the entire network for each DHCP/ARP request, (ii) uses an MPLS header when encapsulating user frames, so that it works with heterogeneous technologies, including those that do not use IEEE 802 addresses, and (iii) offers a LAN-like service model to support all 802.11-based end user mobile terminals;

   (b) An associated routing protocol — WMRP — that supports mobility of both Rbridges and mobile terminals and distributes the IP/MAC associations needed for the data plane ARP optimizations. This protocol is a simple link-state routing protocol with two main differences: (1) the rate of periodic messages is much lower than usual, (2) additional message types are defined for disseminating IP and MAC address information;

   (c) Two competing WMRP optimizations that allow fast dissemination of mobile terminals' location changes (fast handover) using only a residual routing overhead, but without needing location information (GPS) or any kind of network segmentation or hierarchy;

   While maintaining relatively low architectural complexity, by not requiring a GPS sensor, we have demonstrated via simulation that WiMetroNet scales better than existing solutions;

2. A new unified simulation/implementation protocol development methodology that takes advantage of the existing ns-3 network emulation functionality. This contribution comprises an evaluation of the packet processing performance, in terms of achievable throughput, packet loss, and round-trip time, of ns-3 working in emulation mode, compared to a pure kernelspace IPv4 forwarding, and also numerous ns-3 improvements to make developing new protocols easier, including:

(a) Proposal of additional ns-3 classes that significantly improve the emulation of control plane protocols, and simplify the deployment of such protocols;

(b) New visualization capabilities for ns-3 that help solve some problems that occur during protocol development, codenamed *PyViz*. The main innovative aspect of PyViz is that it visualizes a running simulation, instead of a postmortem trace file, and allows researchers to experiment in modifying the simulation state. For instance, it is possible to drag a wireless node with the mouse, while the simulation runs, which is much faster to do than programming a test mobility model in the simulation script;

(c) A new scripting framework for ns-3, based on Python. Unlike ns-2's oTCL bindings, the new Python bindings are strictly a layer on top of the pure C++ simulator, and therefore completely optional. Only objects explicitly exposed by the simulation script consume the associated memory overhead, while ns-3 objects that remain hidden from the simulation script do not incur any memory overhead due to the Python bindings;

(d) New data collection framework (Flow Monitor) for ns-3. The Flow Monitor collects statistics of packet flows passing in simulated nodes, avoiding the need to collect large ascii or pcap trace files, thus speeding up the simulation considerably. It is also simple to use, since it requires nearly no configuration, and is based on an open and modular architecture, allowing it to be extended to classify flows differently and from different sources.

The main innovation in the proposed protocol development methodology is derived by the ease of development afforded by the ns-3 simulator, improved by the emulation performance enhancement that we propose. Moreover, the combination of novel visualization, efficient Python scripting layer, and easy to use data collection framework allows for faster and more consistent protocol development than was previously possible.

5.3 The SITMe project

At the time of this writing, a research project called SITMe (Serviços Integrados para Transportes Metropolitanos) is active. The main goal of this project is to develop and supply information services to passengers traveling

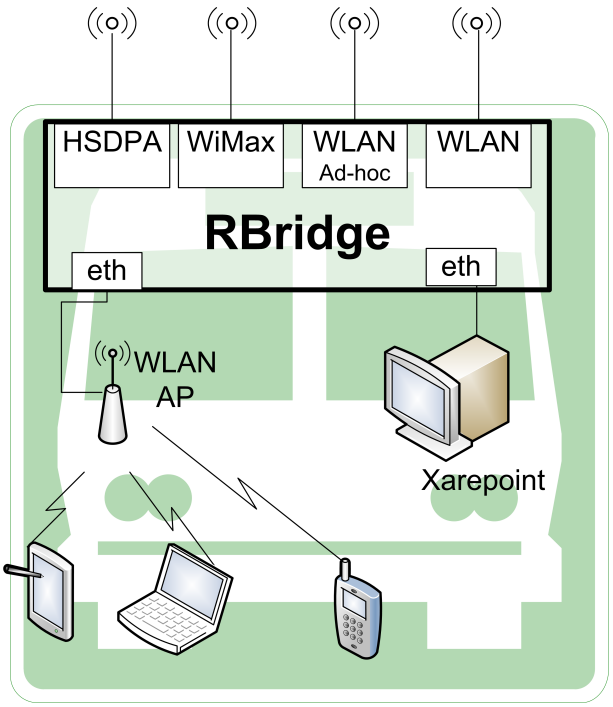


Figure 5.1: A SITMe Rbridge

in buses, such as (1) display news and entertainment information, (2) supply inter-modality information (passenger travel assistance), (3) interactive services, and (4) supply Internet access to passengers using the bus' internal wireless network. For that purpose, a multi-technology communications system is being developed, which is based on the WiMetroNet architecture described in this thesis, with some modifications to accommodate project-specific requirements. In this project, a prototype Rbridge was built, and is being deployed in a small network of eleven buses in the Porto city, Portugal, comprising a bus line near the Faculty of Engineering of the Porto University. The partners of the SITMe project are: **INESC TEC**, **Xarevision**, **FEUP**, and **FEP**. Additionally, project trials are being done in **STCP** buses, with access to a WiMax network being provided by **ONI**, and access to a public IEEE 802.11 hot-spot network provided by **Porto Digital**.

As shown in Fig. 5.1, a SITMe Rbridge contains a total of six network interfaces. Two of those interfaces are internal Ethernet interfaces: one connects to a 802.11 Access Point that provides connectivity to passengers in the bus, while the other connects to the Xarepoint, a set-top box that

controls the information display, developed by Xarevision. The remaining four network interfaces are used to connect the Rbridge to the outside world. Here, never the term "heterogeneous networking" applied better. Currently, one UMTS (HSDPA) card provides Internet connectivity via a traditional cellular network operator. There is also a WiMax card; it will connect to a WiMax base station installed specifically for the SITMe project. One WiFi card, working in *infrastructure mode*, allows the Rbridge to access public WiFi hotspots, in particular the "Porto Digital" network of public WiFi hotspots. Finally, an additional WiFi card, this one in *adhoc mode*, allows vehicles to exchange traffic directly.

Fig. 5.2 shows the actual hardware used. On the left-hand side is the Xarepoint, on the right-hand side is the Rbridge. In Fig. 5.3 we see the hardware mounted in an actual bus, in the upper cabinet area, with connected monitor and keyboard for configuration and debugging access. The hardware used in the Rbridge is the actual hardware that was used to evaluate the ns-3 emulation performance back in Sec. 4.4.2, namely a mini-itx Intel Atom D510. This hardware was selected due to a number of advantages. It is much less expensive than a "regular" computer system, but offers considerable greater performance than the typical CPU found in wireless access points, as evidenced by the results in Fig. 3.33 on page 113. The small form factor, passive cooling, and Solid State Drive (SSD) storage, are additional features of the hardware that make it suitable for deployment in buses.

Running on the hardware described above, the Rbridge communications software is actually the WiMetroNet simulation model developed in the context of this thesis, and used to obtain the simulation results in Chap. 3. To allow this, we put to practice the emulation techniques described in Sec. 4.4 (page 164).

5.4 Future Work

5.4.1 WiMetroNet

Future work of the WiMetroNet architecture will include measuring the performance of our optimizations under different types of traffic. We are particularly interested in evaluating how the *bindupdate* solution compares to *explosive* as the number of peers increases, in a peer-to-peer application. We will also be designing and evaluating further optimizations, this time to reduce the routing overhead incurred due to mobility of Rbridges themselves (e.g. handover of a bus from one base station to another one). Further work involves optimizing the network "bootstrap" issues when very long

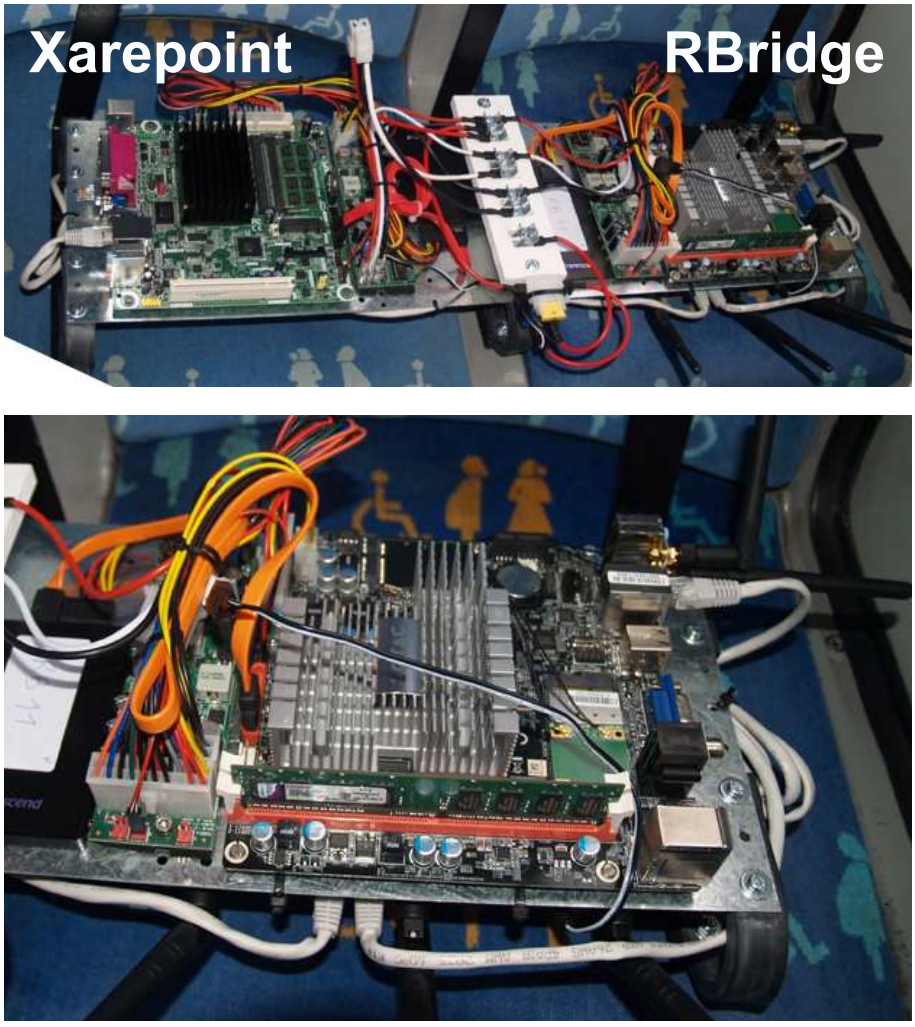


Figure 5.2: SITMe real equipment photo

Figure 5.3: SITMe equipment mounted in a bus

MC/TC/IC refresh intervals are used. Handling temporary disconnection of an Rbridge, which may lead to loss of a periodic global update, will also be improved.

There may be multiple wireless networks available in to be used, but some of them are expensive (3G), and some of them are unreliable or only temporarily available, such as public WiFi networks. The WiMetroNet takes advantage of all heterogeneous networking capabilities available in the region, but suffers from poor reliability of some WiFi APs. One option of adding reliability is to have multipath communications, for example by using both WiFi and UMTS at the same time, duplicating all packets for both networks. Moreover, intelligence is required to try to predict whether a given WiFi AP will be reliable or not, and whether it is worth connecting to it given the coverage area and vehicle trajectory. The use of Machine Learning and Swarm Intelligence techniques applied to this problem will be studied.

5.4.2 Protocol development

Future work on the topic of ns-3 will include better build system control to disable ns-3 modules that are not needed, to save memory, but without needing to modify the build scripts manually. Another area for improvement is to create additional socket classes, similar to RealUdpSocket, for other types of sockets, such as PacketSocket (for layer 2 protocols) and TcpSocket (for TCP protocols). Additionally, a sequence diagram generator, using simulation packet traces as input, would be extremely helpful to protocol developers.

There is also room for improvement in the Flow Monitor ns-3 module. For instance, more data output methods, such as database and binary file, would be welcome. Another useful addition could be to create more options to better control what level of detail is stored in memory. For instance, we might not be interested in per-probe flow statistics, or in histograms. FlowMonitor support for multicast/broadcast flows is another features that could be useful for certain researchers. The FlowMonitor could benefit from a closer integration with NetDevices, e.g. so that it could directly monitor packet drop events from each NetDevice's transmission queue, as well as handle transmission errors from layer 2, instead of relying on the vaguer "packet missing in action" measurement method. In some circumstances a researcher might want to observe how flow metrics evolve over time, instead of just obtaining a summary of the results over the entire simulation. This would require saving a periodic snapshot of the flows metrics to a file. It

can be done already by user code, but a convenience API for it would be interesting to have in FlowMonitor. Finally, we would like to add convenient methods to the Histogram class that would compute the values N, μ, and s that were mentioned in Sec. 4.2.1.

Possible visualization improvements include the ability to visualize layers or objects inside each node, as well visualize the data flows between those objects. It would also be useful to be able to see messages exchanged between objects. Another area in which a graphical visualizer could help would be to automatically generate message sequence diagrams between nodes or objects within each node. Finally, adding an optional "offline" mode to visualization, as done in the traditional visualizers, could be useful for simulations that are too computationally intensive to be able to visualize in real time.

Abbreviations

ACM Association for Computing Machinery (43)

ALPINE Application Level Protocol Infrastructure for Network Experimentation (184 185)

AODV Ad hoc On Demand Distance Vector (2 20 25 53 55 58 116 118 181 189)

AOP Aspect-oriented Programming (85)

AP Access Point (1 9 13 35 63 65 67 74 80 95 116 198)

API Application Programming Interface (47 49 51 123 125 127 131 135 158 160 161 175 183 185 186 191 199)

AR Access Router (35 37 39 55 190)

ARP Address Resolution Protocol (2 5 58 63 70 74 78 82 115 116 118 190 192)

BA Binding Update (94)

BGP Border Gateway Protocol (16)

BSD Berkeley Software Distribution (184)

BSS Basic Service Set (10 13 23)

BSSID BSS Identifier (13)

BU Binding Update (88 94)

CBR Constant Bit-rate (96 146)

CCH Control Channel (12)

CN Correspondent Node (29 32 37 39 55 189)

COM Component Object Model (47)

CPU Central Processing Unit (52 113 137 153 165 171 173 177 178 195)

CSMA Carrier Sense Multiple Access (8 47 49 53)

CTS Clear To Send (12)

DB Data Base (154)

DBMS Data Base Management System (153 154)

DCF Distributed Coordination Function (9 12)

DDR Double Data Rate (113)

DHCP Dynamic Host Configuration Protocol (2 7 33 37 54 57 59 61 63 64 68 76 86 115 116 118 189 190 192)

DNS Domain Name System (37)

DNS-SD DNS based Service Discovery (115)

DS Distribution System (10 23)

DSCP Differentiated Services Code Point (140)

DSR Dynamic Source Routing (53)

DSRC Dedicated Short-Range Communications (8 12)

EMM Enhanced Mobility Management (117)

ESS Extended Service Set (10 25)

ETX Expected Transmission Count (182)

EUI Extended Unique Identifier (48 54 61)

FA Foreign Agent (29 32)

FBU Fast Binding Update (35)

FCS Frame Check Sequence (8)

FEC Forward Error Correction (21 82)

FEP Faculdade de Economia da Universidade do Porto (194)

FEUP Faculdade de Engenharia da Universidade do Porto (194)

FIB Forwarding Information Base (16)

FNA Fast Neighbor Advertisement (35)

FTP File Transfer Protocol (174)

GCC GNU Compiler Collection (133)

GFA Gateway Foreign Agent (32)

GIL Global Interpreter Lock (139)

GPL GNU General Public License (56 190)

GPS Global Positioning System (116 192)

GTK Gimp Tool Kit (154)

GUI Graphical User Interface (139 154 157)

GW Gateway (67 68)

HA Home Agent (29 32 35)

HDF Hierarchical Data Format (138)

HI Handover Initiate (35 37)

HIP Host Identity Protocol (37 38 41 55 189)

HIT Host Identity Tag (37)

HNA Host and Network Association (18 64 99 117)

HSDPA High-Speed Downlink Packet Access (10 13 195)

HTTP Hyper Text Transfer Protocol (174)

HWMP Hybrid Wireless Mesh Protocol (25 116)

IC IP Control (60 61 63 70 76 86 198)

ICMP Internet Control Message Protocol (47)

IEEE Institute of Electrical and Electronics Engineers (1 8 9 11 14 48 54 64 115 118 192 194)

IETF Internet Engineering Task Force (22 28 114 144)

IP Internet Protocol (1 5 7 11 14 16 17 22 23 26 27 28 28 31 33 35 36 39 39 42 44 47 55 56 57 60 61 64 65 69 76 82 118 137 141 146 171 172 177 182 189 190 192)

IS-IS Intermediate System To Intermediate System (22)

ISM Industrial, Scientific and Medical band (9 12)

LAN Local Area Network (2 4 9 16 23 28 55 177 189 192)

LANMAR Landmark Routing Protocol for Large Scale Networks (115)

LAR Location-Aided Routing (116)

LLC Logical Link Control (15)

LMA Local Mobility Anchor (33)

LSP Label Switched Path (21 68)

LTA Local Terminal Associations (63 71)

LTE Long Term Evolution (10)

MAC Media Access Control (5 8 9 11 12 15 22 25 26 33 44 48 51 54 59 60 61 63 65 67 68 70 71 74 76 80 82 82 93 94 95 101 113 118 146 150 175 182 186 190 192)

MAG Mobile Access Gateways (33)

MAMP Mobility-Aware Multi-Path (116)

MAP Mesh Access Point (116)

MC MAC Control (25 60 61 65 71 73 76 86 87 89 90 92 93 94 96 99 101 106 107 111 198)

MID Multiple Interface Declaration (18)

MIP Mobile IP (29 31 34 35 41 41 56 190)

MIP-RR Mobile Internet Protocol Regional Registration (32)

MIPS Microprocessor without Interlocked Pipeline Stages (113)

MN Mobile Node (29 31 32 34 35 37 39)

MP Mesh Point (116)

MPLS Multi-protocol Label Switching (14 21 26 27 55 59 60 63 68 72 75 76 78 81 82 84 88 92 118 139 171 189 190 192)

MPP Mesh Portal Points (116)

MPR Multi-point Relay (18 19 64 66)

MSC Message Sequence Chart (72)

MTU Maximum Transmission Unit (35 171 172 174)

NAR New Access Router (35)

NEMO Network Mobility (34 42)

NSC Network Simulation Cradle (184 185)

OBU Onboard Units (12)

OLSR Optimized Link State Routing (2 17 19 26 47 55 58 60 64 65 86 90 96 99 108 115 117 118 146 153 161 174 176 177 179 181 186 189 190)

OS Operating System (156 182)

OSI Open Systems Interconnection (41)

OSPF Open Shortest Path First (16 19 23 42 189)

PAN Personal Area Network (34)

PAR Previous Access Router (35)

PCF Point Coordination Function (10)

PDP Packet Data Protocol (10)

PDU Protocol Data Unit (37 60 78 80 82 166)

PKI Public Key Infrastructure (37)

PMLAR Predictive Mobility and Location-Aware Routing (116)

PPP Point-to-point Protocol (11 49 150)

PREP Path Response (25)

PREQ Path Request (25)

RANN Root Announcement (25)

RERR Route Error (21)

RFC Request For Comment (59 144)

RID Rbridge Identifier (59 87)

RREP Route Reply (20)

RREQ Route Request (20)

RRM Radio Resource Manager (10)

RSS Resident Memory Size (179)

RSU Roadside Unit (12 13)

RSVP-TE Resource Reservation Protocol – Traffic Engineering (59)

RTA Remote Terminal Associations (63 68)

RTP Real-time Protocol (45)

RTS Request To Send (12)

RTT Round-trip Time (171 172)

SCH Service Channel (12)

SDL Specification and Description Language (53 56)

SIGCOMM Special Interest Group on Data Communications (157)

SIP Session Initiation Protocol (39 39 41 55 124 131 133 135 189 191)

SLA Service Level Agreement (136)

SNMP Simple Network Management Protocol (136)

SQL Simple Query Language (138 153)

SSD Solid State Drive (195)

SSID Service Set Identifier (64 155)

STCP Sociedade de Transportes Colectivos do Porto (194)

STL Standrd Template Library (125 182)

STP Spanning Tree Protocol (16 54 189)

SVG Scalable Vector Graphics (160)

SWANS Scalable Wireless Ad hoc Network Simulator (53 56 190)

SWIG Simplified Wrapper and Interface Generator (124 131 133 135 191)

TC Topology Control (18 20 60 61 86 99 114 198)

TCL Tool Command Language (44 45 47 122 134)

TCP Transmission Control Protocol (41 43 47 48 96 99 107 110 141 156 171 174 178 198)

TRILL Transparent Interconnection of Lots of Links (22 26 28 55 114 118 189)

TTL Time To Live (16 20 22 25 27 28 59 68 87 94 96 114 144 183)

U-NII Unlicensed National Information Infrastructure (9 13)

UCLA University of California, Los Angeles (52)

UDP User Datagram Protocol (41 48 96 99 107 109 141 146 150 171 175 177 186 191)

UML Unified Modelling Language (74)

UMTS Universal Mobile Telecommunications System (1 8 10 13 54 56 59 189 195 198)

URI Uniform Resource Identifier (39)

URL Uniform Resource Locator (175)

VANET Vehicular Ad-Hoc Network (2)

WAVE Wireless Access for Vehicular Environments (12)

WCDMA Wideband Code Division Multiple Access (10)

WLAN Wireless Local Area Network (1 9 116)

WMAN Wireless Metropolitan Area Network (1 11)

WMN Wireless Mesh Network (115)

WMRP Wireless Metropolitan Routing Protocol (5 59 60 61 64 65 68 74 76 78 80 82 86 86 90 93 94 96 99 108 108 114 118 192)

WNMT Wireless Network for Metropolitan Transports (2 27 57)

XAV XML tracing framework of Yavista (153 154)

XML Extensible Markup Language (138 147 152 154)

ZRP Zone Routing Protocol (54)

Bibliography

[1] M. Ricardo, G. Carneiro, P. Fortuna, F. Abrantes, and J. Dias, "WiMetroNet — a scalable wireless network for metropolitan transports," *Advanced International Conference on Telecommunications*, vol. 0, pp. 520–525, 2010.

[2] A. Myers, E. Ng, and H. Zhang, "Rethinking the service model: Scaling ethernet to a million nodes," 2004. [Online]. Available: citeseer.ist.psu.edu/myers04rethinking.html

[3] T. Clausen and P. Jacquet, "Optimized Link State Routing Protocol (OLSR)," RFC 3626 (Experimental), Internet Engineering Task Force, Oct. 2003. [Online]. Available: http://www.ietf.org/rfc/rfc3626.txt

[4] C. Perkins, E. Belding-Royer, and S. Das, "Ad hoc On-Demand Distance Vector (AODV) Routing," RFC 3561 (Experimental), Internet Engineering Task Force, Jul. 2003. [Online]. Available: http://www.ietf.org/rfc/rfc3561.txt

[5] R. Metcalfe and D. Boggs, "Ethernet: Distributed packet switching for local computer networks," *Communications of the ACM*, vol. 19, no. 7, pp. 395–404, 1976.

[6] G. Hiertz, D. Denteneer, L. Stibor, Y. Zang, X. Costa, and B. Walke, "The IEEE 802.11 universe," *Communications Magazine, IEEE*, vol. 48, no. 1, pp. 62–70, 2010.

[7] P. Mogensen, T. Koivisto, K. Pedersen, I. Kovács, B. Raaf, K. Pajukoski, and M. Rinne, "LTE-Advanced: The path towards gigabit/s in wireless mobile communications," in *Wireless Communication, Vehicular Technology, Information Theory and Aerospace & Electronic Systems Technology, 2009. Wireless VITAE 2009. 1st International Conference on.* IEEE, 2009, pp. 147–151.

[8] "IEEE Std 802.16e-2005 and IEEE Std 802.16-2004/Cor 1-2005: IEEE standard for local and metropolitan area networks part 16: Air interface for fixed and mobile broadband wireless access systems amendment 2: Physical and medium access control layers for combined fixed and mobile operation in licensed bands and corrigendum 1," Tech. Rep., 2006. [Online]. Available: http://dx.doi.org/10.1109/IEEESTD.2006.99107

[9] P. Grønsund, P. Engelstad, and M. A. T. Skeie, *Real Life Field Trial over a Pre-mobile WiMAX System with 4th Order Diversity*, ser. Lecture Notes in Computer Science. Springer, August 2007, vol. 4712/2007.

[10] D. Jiang, V. Taliwal, A. Meier, W. Holfelder, and R. Herrtwich, "Design of 5.9 GHz DSRC-based vehicular safety communication," *Wireless Communications, IEEE*, vol. 13, no. 5, pp. 36–43, 2006.

[11] J. Kenney, "Dedicated Short-Range Communications (DSRC) standards in the United States," *Proceedings of the IEEE*, vol. 99, no. 7, pp. 1162–1182, 2011.

[12] N. Balasubramanian, A. Balasubramanian, and A. Venkataramani, "Energy consumption in mobile phones: a measurement study and implications for network applications," in *Proceedings of the 9th ACM SIGCOMM conference on Internet measurement conference*. ACM, 2009, pp. 280–293.

[13] M. Bhatia *et al.*, *Introduction to Computer Network*. Madhulika, 2009.

[14] J. Moy *et al.*, "OSPF version 2," STD 54, RFC 2328, April, Tech. Rep., 1998.

[15] Y. Rekhter, T. Li, and S. Hares, "A Border Gateway Protocol (BGP-4)," RFC 4271, January, Tech. Rep., 2006.

[16] E. Dijkstra, "A note on two problems in connexion with graphs," *Numerische mathematik*, vol. 1, no. 1, pp. 269–271, 1959.

[17] J. P. Macker and J. W. Dean, *A Study of Link State Flooding Optimizations for Scalable Wireless Networks*. Storming Media, 2003.

[18] L. Viennot, L. Viennot, and P. Hipercom, "Complexity results on election of multipoint relays in wireless networks," *INTERNAL*

REPORT RR-3584, INRIA, 1998. [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.43.4536

[19] O. Liang, Y. A. Sekercioglu, and N. Mani, "A survey of multipoint relay based broadcast schemes in wireless ad hoc networks," *IEEE Communications Surveys & Tutorials*, vol. 8, no. 4, pp. 30–46, 2006.

[20] C. Adjih, E. Baccelli, T. Clausen, P. Jacquet, and G. Rodolakis, "Fish eye olsr scaling properties," *IEEE Journal of Communication and Networks (JCN), Special Issue on Mobile Ad Hoc Wireless Networks*, vol. 6, no. 4, pp. 343–351, 2004.

[21] J. Härri, C. Bonnet, and F. Filali, "OLSR and MPR: mutual dependences and performances," in *Challenges in Ad Hoc Networking*, ser. IFIP International Federation for Information Processing. Springer Boston, 2006, pp. 67–71.

[22] A. Busson, N. Mitton, and E. Fleury, "Analysis of the multi-point relay selection in olsr and implications," in *Challenges in Ad Hoc Networking: Fourth Annual Mediterranean Ad Hoc Networking Workshop, June 21-24, 2005, Île de Porquerolles, France*. Springer, 2006, p. 387.

[23] E. Rosen, A. Viswanathan, and R. Callon, "Rfc3031: Multiprotocol label switching architecture," *Internet RFCs*, 2001.

[24] J. Touch and R. Perlman, "Transparent Interconnection of Lots of Links (TRILL): Problem and Applicability Statement," RFC 5556 (Informational), Internet Engineering Task Force, May 2009. [Online]. Available: http://www.ietf.org/rfc/rfc5556.txt

[25] A. INCITS, "Iso/iec 10589: 2002," *Information technology–Telecommunications and information exchange between systems–Intermediate System to intermediate system intra-domain routeing information exchange protocol for use in conjunction with the protocol for providing the connectionless-mode network service (ISO 8473)*.

[26] "IEEE Draft Standard for Information Technology-Telecommunications and information exchange between systems-local and metropolitan area networks-specific requirements-part 11: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) specifications-Amendment 10: Mesh Networking," *IEEE Std P802.11s/D5.0*, April 2010.

[27] N. Wangi, R. Prasad, M. Jacobsson, and I. Niemegeers, "Address autoconfiguration in wireless ad hoc networks: protocols and techniques," *Wireless Communications, IEEE*, vol. 15, no. 1, pp. 70–80, 2008.

[28] C. Perkins, "IP Mobility Support for IPv4, Revised," RFC 5944 (Proposed Standard), Internet Engineering Task Force, Nov. 2010. [Online]. Available: http://www.ietf.org/rfc/rfc5944.txt

[29] C. Perkins, D. Johnson, and J. Arkko, "Mobility Support in IPv6," RFC 6275 (Proposed Standard), Internet Engineering Task Force, Jul. 2011. [Online]. Available: http://www.ietf.org/rfc/rfc6275.txt

[30] E. Fogelstroem, A. Jonsson, and C. Perkins, "Mobile IPv4 Regional Registration," RFC 4857 (Experimental), Internet Engineering Task Force, Jun. 2007. [Online]. Available: http://www.ietf.org/rfc/rfc4857.txt

[31] S. Gundavelli, K. Leung, V. Devarapalli, K. Chowdhury, and B. Patil, "Proxy Mobile IPv6," RFC 5213 (Proposed Standard), Internet Engineering Task Force, Aug. 2008. [Online]. Available: http://www.ietf.org/rfc/rfc5213.txt

[32] R. Wakikawa and S. Gundavelli, "IPv4 Support for Proxy Mobile IPv6," RFC 5844 (Proposed Standard), Internet Engineering Task Force, May 2010. [Online]. Available: http://www.ietf.org/rfc/rfc5844.txt

[33] K. Leung, G. Dommety, V. Narayanan, and A. Petrescu, "Network Mobility (NEMO) Extensions for Mobile IPv4," RFC 5177 (Proposed Standard), Internet Engineering Task Force, Apr. 2008. [Online]. Available: http://www.ietf.org/rfc/rfc5177.txt

[34] C. Ng, P. Thubert, M. Watari, and F. Zhao, "Network Mobility Route Optimization Problem Statement," RFC 4888 (Informational), Internet Engineering Task Force, Jul. 2007. [Online]. Available: http://www.ietf.org/rfc/rfc4888.txt

[35] R. Koodli, "Mobile IPv6 Fast Handovers," RFC 5568 (Proposed Standard), Internet Engineering Task Force, Jul. 2009. [Online]. Available: http://www.ietf.org/rfc/rfc5568.txt

[36] R. Koodli and C. Perkins, "Mobile IPv4 Fast Handovers," RFC 4988 (Experimental), Internet Engineering Task Force, Oct. 2007. [Online]. Available: http://www.ietf.org/rfc/rfc4988.txt

[37] K. E. Malki, "Low-Latency Handoffs in Mobile IPv4," RFC 4881 (Experimental), Internet Engineering Task Force, Jun. 2007. [Online]. Available: http://www.ietf.org/rfc/rfc4881.txt

[38] R. Moskowitz and P. Nikander, "Host Identity Protocol (HIP) Architecture," RFC 4423 (Informational), Internet Engineering Task Force, May 2006. [Online]. Available: http://www.ietf.org/rfc/rfc4423.txt

[39] R. Moskowitz, P. Nikander, P. Jokela, and T. Henderson, "Host Identity Protocol," RFC 5201 (Experimental), Internet Engineering Task Force, Apr. 2008, updated by RFC 6253. [Online]. Available: http://www.ietf.org/rfc/rfc5201.txt

[40] P. Jokela, T. Rinta-aho, T. Jokikyyny, J. Wall, M. Kuparinen, H. Mahkonen, J. Melén, T. Kauppinen, and J. Korhonen, "Handover performance with HIP and MIPv6," in *Wireless Communication Systems, 2004, 1st International Symposium on*. IEEE, 2004, pp. 324–328.

[41] J. Rosenberg, H. Schulzrinne, G. Camarillo, A. Johnston, J. Peterson, R. Sparks, M. Handley, and E. Schooler, "SIP: Session Initiation Protocol," RFC 3261 (Proposed Standard), Internet Engineering Task Force, Jun. 2002, updated by RFCs 3265, 3853, 4320, 4916, 5393, 5621, 5626, 5630, 5922, 5954, 6026, 6141. [Online]. Available: http://www.ietf.org/rfc/rfc3261.txt

[42] C. Perkins, "Mobile IP," *Communications Magazine, IEEE*, vol. 40, no. 5, pp. 66–82, May 2002.

[43] S. Gundavelli, K. Leung, V. Devarapalli, and K. C. B. Patil, "Proxy mobile ipv6," Internet draft, IETF, draft-ietf-netlmm-proxymip6-07.txt, Nov. 2007.

[44] G. Fishman, *Discrete-event simulation: modeling, programming, and analysis*. Springer Verlag, 2001.

[45] S. Kurkowski, T. Camp, and M. Colagrosso, "Manet simulation studies: The current state and new simulation tools," *Mobile Computing and Communications Review*, vol. 9, no. 4, pp. 50–61, 2005.

[46] D. Box, *Essential COM*. Addison-Wesley, 1998.

[47] E. Weingärtner, H. vom Lehn, and K. Wehrle, "A performance comparison of recent network simulators," in *Proceedings of the IEEE International Conference on Communications 2009 (ICC 2009), Dresden, Germany, IEEE.*, Jun. 2009.

[48] J. Font, P. Inigo, M. Dominguez, J. Sevillano, and C. Amaya, "Analysis of source code metrics from ns-2 and ns-3 network simulators," *Simulation Modelling Practice and Theory*, 2011.

[49] D. Edelson, *Smart pointers: They're smart, but they're not pointers.* University of California, Santa Cruz, Computer Research Laboratory, 1992.

[50] T. Camp, J. Boleng, and V. Davies, "A survey of mobility models for ad hoc network research," *Wireless Communications and Mobile Computing*, vol. 2, no. 5, pp. 483–502, 2002.

[51] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design patterns: elements of reusable object-oriented software.* Addison-Wesley, 1995.

[52] G. Carneiro, J. Ruela, and M. Ricardo, "Cross-layer design in 4G wireless terminals," *Wireless Communications, IEEE [see also IEEE Personal Communications]*, vol. 11, no. 2, pp. 7–13, 2004.

[53] A. Varga and R. Hornig, "An overview of the OMNeT++ simulation environment," in *Proceedings of the 1st international conference on Simulation tools and techniques for communications, networks and systems & workshops.* ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), 2008, pp. 1–10.

[54] R. Bagrodia, R. Meyer, M. Takai, Y. Chen, X. Zeng, J. Martin, and H. Song, "Parsec: A parallel simulation environment for complex systems," *Computer*, vol. 31, no. 10, pp. 77–85, 1998.

[55] I. Specification, "Description Language (SDL). ITU-T Recommendation Z. 100," *International Telecommunication Union, Geneve*, vol. 267, 1992.

[56] R. Barr, Z. Haas, and R. van Renesse, "Jist: An efficient approach to simulation using virtual machines," *Software: Practice and Experience*, vol. 35, no. 6, pp. 539–576, 2005.

[57] R. Barr, "Swans-scalable wireless ad hoc network simulator user guide," 2006.

[58] C. Rigney, S. Willens, A. Rubens, and W. Simpson, "Remote Authentication Dial In User Service (RADIUS)," RFC 2865, IETF, 2000.

[59] "IEEE standard for local and metropolitan area networks port-based network access control," *IEEE Std 802.1X-2004 (Revision of IEEE Std 802.1X-2001)*, 2004.

[60] J. Daemen and V. Rijmen, *The Design of Rijndael: AES - The Advanced Encryption Standard*. Springer, 2002.

[61] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *Communications of the ACM*, vol. 21, no. 7, pp. 558–565, 1978.

[62] T. Elrad, R. Filman, and A. Bader, "Aspect-oriented programming: Introduction," *Communications of the ACM*, vol. 44, no. 10, pp. 29–32, 2001.

[63] D. Kim, H. Cai, M. Na, and S. Choi, "Performance measurement over mobile WiMAX/IEEE 802.16 e network," in *World of Wireless, Mobile and Multimedia Networks, 2008. WoWMoM 2008. 2008 International Symposium on a*, 2008, pp. 1–8.

[64] Hongqiang Zhai, Younggoo Kwon, and Yuguang Fang, "Performance analysis of IEEE 802.11 MAC protocols in wireless LANs," *Wireless Communications and Mobile Computing*, vol. 4, no. 8, pp. 917–931, 25 Nov. 2004.

[65] N. Potnis and A. Mahajan, "Mobility models for vehicular ad hoc network simulations," in *Proceedings of the 44th annual Southeast regional conference*. ACM, 2006, p. 747.

[66] C. Demetrescu and G. Italiano, "Experimental analysis of dynamic all pairs shortest path algorithms," *ACM Transactions on Algorithms (TALG)*, vol. 2, no. 4, pp. 578–601, 2006.

[67] G. Pei, M. Gerla, and X. Hong, "LANMAR: landmark routing for large scale wireless ad hoc networks with group mobility," in *Proceedings of the 1st ACM international symposium on Mobile ad hoc networking & computing*, 2000, p. 18.

[68] F. J. Ros and P. M. Ruiz, "Cluster-based OLSR extensions to reduce control overhead in mobile ad hoc networks," in *Proceedings of the*

2007 international conference on Wireless communications and mobile computing, 2007, p. 207.

[69] Y. Ko and N. Vaidya, "Location-aided routing (LAR) in mobile ad hoc networks," *Wireless Networks*, vol. 6, no. 4, p. 321, 2000.

[70] K. FENG, C. HSU, and T. LU, "Velocity-assisted predictive mobility and location-aware routing protocols for mobile ad hoc networks," *IEEE transactions on vehicular technology*, vol. 57, no. 1, pp. 448–464, 2008.

[71] T. Taleb, E. Sakhaee, A. Jamalipour, K. Hashimoto, N. Kato, and Y. Nemoto, "A stable routing protocol to support ITS services in VANET networks," *IEEE Transactions on Vehicular Technology*, vol. 56, no. 6 Part 1, pp. 3337–3347, 2007.

[72] V. Namboodiri and L. Gao, "Prediction-based routing for vehicular ad hoc networks," *IEEE Transactions on Vehicular Technology*, vol. 56, no. 4 Part 2, pp. 2332–2345, 2007.

[73] Y. Fan, J. Zhang, and X. Shen, "Mobility-aware multi-path forwarding scheme for wireless mesh networks," in *Wireless Communications and Networking Conference, 2008. WCNC 2008. IEEE.* IEEE, 2008, pp. 2337–2342.

[74] A. Capone, S. Napoli, and A. Pollastro, "Mobimesh: an experimental platform for wireless mesh networks with mobility support," in *Proc. of ACM QShine 2006 Workshop on "Wireless mesh: moving towards applications", Waterloo (Canada)*, 2006.

[75] H. Wang, Q. Huang, Y. Xia, Y. Wu, and Y. Yuan, "A network-based local mobility management scheme for wireless mesh networks," in *Wireless Communications and Networking Conference, 2007. WCNC 2007. IEEE.* IEEE, 2007, pp. 3792–3797.

[76] Y. Amir, C. Danilov, M. Hilsdale, R. Musaloiu-Elefteri, and N. Rivera, "Fast handoff for seamless wireless mesh networks," in *Proceedings of the 4th international conference on Mobile systems, applications and services.* ACM, 2006, pp. 83–95.

[77] M. Bezahaf, L. Iannone, and S. Fdida, "Enhanced mobility management in wireless mesh networks," *Journées Doctorales em Informatique et Réseaux (JDIR08)*, 2008.

[78] L. Couto, J. Barraca, S. Sargento, and R. Aguiar, "FastM in WMN: A Fast Mobility Support Extension for Wireless Mesh Networks," in *2009 Second International Conference on Advances in Mesh Networks*. IEEE, 2009, pp. 90–96.

[79] A. Roos, A. Roos, M. Flegl, S. Wieland, N. Bayer, D. Sivchenko, J. Habermann, P. Behbahani, V. Rakocevic, A. T. Schwarzbacher, B. Xu, G. Zimmermann, and G. Kadel, "Broadband wireless internet access in public transportation," in *Proceedings of VDE-Kongress 2006 - Innovations for Europe*, 2006.

[80] S. W. Smith, *The Scientist & Engineer's Guide to Digital Signal Processing*, 1st ed. California Technical Pub., 1997.

[81] J. Malek and K. Nowak, "Trace graph-data presentation system for network simulator ns," in *Proceedings of the Information Systems - Concepts, Tools and Applications (ISAT 2003)*, Poland, September 2003.

[82] R. Ben-El-Kezadri, F. Kamoun, and G. Pujolle, "XAV: a fast and flexible tracing framework for network simulation," in *Proceedings of the 11th international symposium on Modeling, analysis and simulation of wireless and mobile systems*. Vancouver, British Columbia, Canada: ACM, 2008, pp. 47–53. [Online]. Available: http://portal.acm.org/citation.cfm?id=1454515

[83] C. Cicconetti, E. Mingozzi, and G. Stea, "An integrated framework for enabling effective data collection and statistical analysis with ns-2," in *Proceeding from the 2006 workshop on ns-2: the IP network simulator*. Pisa, Italy: ACM, 2006, p. 11. [Online]. Available: http://portal.acm.org/citation.cfm?doid=1190455.1190466

[84] J. Ellson, E. Gansner, L. Koutsofios, S. North, and G. Woodhull, "Graphviz—open source graph drawing tools," in *Graph Drawing*. Springer, 2002, pp. 594–597.

[85] S. Kurkowski, T. Camp, N. Mushell, and M. Colagrosso, "A visualization and analysis tool for NS-2 wireless simulations: iNSpect," in *Modeling, Analysis, and Simulation of Computer and Telecommunication Systems, 2005. 13th IEEE International Symposium on*. IEEE, 2005, pp. 503–506.

[86] R. Alur, "Timed automata," in *Computer Aided Verification*, ser. Lecture Notes in Computer Science, N. Halbwachs and D. Peled, Eds. Springer Berlin / Heidelberg, 1999, vol. 1633, pp. 688–688.

[87] D. Couto, D. Aguayo, J. Bicket, and R. Morris, "A high-throughput path metric for multi-hop wireless routing," *Wireless Networks*, vol. 11, no. 4, pp. 419–434, 2005.

[88] R. Morris, E. Kohler, J. Jannotti, and M. Kaashoek, "The Click modular router," in *Proceedings of the seventeenth ACM symposium on Operating systems principles*. ACM, 1999, pp. 217–231.

[89] M. Neufeld, A. Jain, and D. 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*. ACM, 2002, pp. 74–81.

[90] J. Morrison and J. Morrison, *Flow-Based Programming: A new approach to application development*. Van Nostrand Reinhold Princeton, NJ, 1994.

[91] S. Muthukumar, X. Li, C. Liu, J. Kopena, M. Oprea, and B. Loo, "Declarative toolkit for rapid network protocol simulation and experimentation," *SIGCOMM (demo)*, 2009.

[92] M. Fras, G. Globačnik, and J. Mohorko, "Advanced method of network simulations with opnet modeler," in *proceedings of the 14th National Conference on High Education TREND, Kopaonik*, 2008.

[93] X. Huang, R. Sharma, and S. Keshav, "The ENTRAPID protocol development environment," in *INFOCOM'99. Eighteenth Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE*, vol. 3. IEEE, 2002, pp. 1107–1115.

[94] D. Ely, S. Savage, and D. Wetherall, "Alpine: A user-level infrastructure for network protocol development," in *Proceedings of the 3rd conference on USENIX Symposium on Internet Technologies and Systems-Volume 3*. USENIX Association, 2001, p. 15.

[95] S. Jansen and A. McGregor, "Simulation with real world network stacks," in *Proceedings of the 37th conference on Winter simulation*. Winter Simulation Conference, 2005, pp. 2454–2463.

[96] M. Zec and M. Mikuc, "Operating system support for integrated network emulation in IMUNES," in *Proc. of the 1st Workshop on Operating System and Architectural Support for the on demand IT InfraStructure (OASIS), Boston, MA*, 2004.