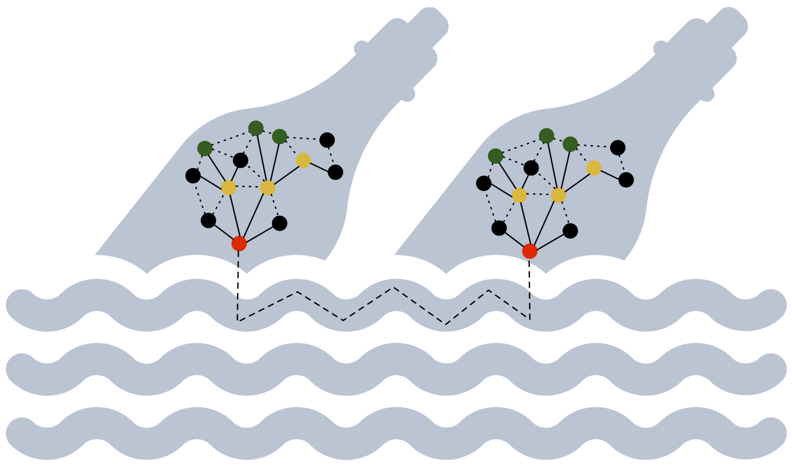
Philipps Universität Marburg

Distributed Simulation and Emulation of Mobile Nodes in Dynamic Networks

MiniWorld
Mobile Infrastructure'n'Network Integrated World



Nils Tobias Schmidt

December 14, 2016

Acknowledgements

Writing a thesis is not an easy job. Thankfully, many people supported me during that period. Special thanks go to **Svenja Seip**, my beloved girlfriend, which always had understanding for my enthusiasm while coding and writing this thesis. She supported me at home and provided me every day with new ambition to get the best out of the proposed solution which is called MiniWorld.
My supervisor **Lars Baumgärtner** had always good ideas and provided me with the required feedback. Feedback came also from my old friends which whom I shared an office for approximately 2 years. Working together resulted in the published paper called *An experimental evaluation of delay-tolerant networking with serval* where MiniWorld has been used as an emulation platform the first time.
Moreover, I want to thank **Pablo Graubner** for his input on frameworks for distributed communication. Based on his suggestions, *ZeroMQ* made it into MiniWorld and replaced a slower communication layer. Finally I want to thank **Patrick Lampe** for contributing the *Movement Patterns* to MiniWorld.
Parts of the MiniWorld icon are from Freepik[1]

[1]`www.flaticon.com`

Abstract

Network is a critical part of software since conditions such as delay, duplicate packets, loss etc. require special treatment by algorithms and applications. Emulating such conditions is crucial since certain errors may arise only under special conditions. Especially routing algorithms, distributed applications and protocols in general, require network topologies with many nodes and mobility involved. Physical testbeds, network simulation and network emulation are approaches to create these special environments.
This thesis presents MiniWorld, a distributed network emulation framework. Flexibility, modularity, exchangeability and transparency are incorporated into the design of MiniWorld. The framework points out its feasibility by including 4 network backends for both wired and wireless communication emulation. Moreover, MiniWorld uses WiFi virtualization to offer KVM based VMs a real WiFi interface. Connection tracking, differential network switching and network checking is provided to each network backend automatically.
A study of 802.11s shows the feasibility of the WiFi virtualization approach. To remove performance bottlenecks, MiniWorld supports distributed emulation with different VM scheduling algorithms. The Serval mesh software is used to show the distributed mode of MiniWorld. 256 KVM VMs are interconnected in a chain topology and distributed across 7 computers.

Deutsche Zusammenfassung

Das Netzwerk einer Applikation ist ein kritischer Bestandteil, da bestimmte Umgebungsvariablen wie Verzögerungen und doppelte, fehlerhafte oder verlorene Pakete eine besondere Behandlung durch Algorithmen und Anwendungen benötigen. Die Emulation solcher Netzwerkbedingungen ist allerdings notwendig, um gerade die Fehler einer Applikation ausfindig zu machen, die nur unter bestimmten Voraussetzungen auftreten. Besonders betroffen davon sind Routing-Algorithmen und verteilte Applikationen, da in diesen Fällen Netzwerk-Topologien mit vielen Knoten und einem hohen Grad an Mobilität benötigt werden. Physikalische Testaufbauten, Netzwerkemulation und Netzwerksimulation sind Ansätze, die versuchen auf diese speziellen Anforderungen einzugehen.
Im Rahmen dieser Thesis wird das verteilte framework für Netzwerkemulation mit dem Namen MiniWorld vorgestellt. Flexibilität, Modularität und Austauschbarkeit sind in das Design von MiniWorld eingeflossen, sodass 4 Netzwerk-Backends sowohl kabelgebundene als auch funkbasierte Kommunikation simulieren können. WiFi Virtualisierung wird benutzt, um WiFi Karten in KVM basierten VMs bereitzustellen. Verbindungen zwischen Knoten werden durch MiniWorld verwaltet, sodass diese automatisch konfiguriert und überprüft werden.
Eine Analyse von 802.11s zeigt die Notwendigkeit des Ansatzes der WiFi Virtualisierung. MiniWorld unterstützt die verteilte Emulation mit unterschiedlichen VM Scheduling-Algorithmen, um Performance Engpässe zu beseitigen. Die DTN Software Serval wird verwendet, um den verteilten Modus von MiniWorld zu demonstrieren. Dabei werden 256 KVM VMs in einer Chain Topologie verbunden und über 7 Computer verteilt.

Contents

Abstract v

Deutsche Zusammenfassung vii

Acknowledgements iv

1 Introduction 1

2 Basics 3
- 2.1 Layered Network Architectures 3
- 2.2 Simulation vs. Testbeds vs. Emulation 4
  - 2.2.1 Network Simulators 4
  - 2.2.2 Testbeds 4
  - 2.2.3 Network Emulators 4
- 2.3 Wireless Technologies 6
  - 2.3.1 IEEE 802 8
- 2.4 Networking in Linux 12
  - 2.4.1 Bridge Netfilter Hooks 13
  - 2.4.2 Traffic Control 14
  - 2.4.3 Linux Wireless Drivers Overview 14
  - 2.4.4 Relation To MiniWorld 16
- 2.5 Virtualization 16
  - 2.5.1 Hypervisor Virtualization 16
  - 2.5.2 Container Virtualization 19

3 Related Work 21
- 3.1 Simulation 21
- 3.2 Network Emulation 22
  - 3.2.1 Netem 22
  - 3.2.2 VDE 22
  - 3.2.3 Open vSwitch 23
- 3.3 WiFi Virtualization 24
  - 3.3.1 Combined Software And Hardware Based Approach 25
  - 3.3.2 Qemu Atheros NIC Model 26
  - 3.3.3 Mac80211_Hwsim & Wmediumd 26
- 3.4 Emulation Systems 27
  - 3.4.1 CORE 27
  - 3.4.2 IMUNES 30
  - 3.4.3 Dockemu 30
  - 3.4.4 Mininet 31
  - 3.4.5 Mininet-WiFi 31
  - 3.4.6 Netkit 32

3.4.7 Cloonix-net . . . 32
3.4.8 GNS3 . . . 33
3.4.9 NEMAN . . . 33
3.5 Summary . . . 34

4 Design **35**
4.1 Short Overview & Design Goals . . . 35
4.2 Architecture . . . 36
4.3 Workflow . . . 37
4.4 Scenario Config . . . 38
4.5 Interfaces . . . 38
4.6 NetworkBackend Communication . . . 39
4.6.1 Simulation Lifecycle . . . 39
4.6.2 Simulation Step . . . 41
4.7 REPLable . . . 42
4.8 Link Quality Models . . . 43
4.9 Movement Patterns . . . 46
4.10 Network Backends . . . 46
4.10.1 VDE . . . 47
4.10.2 Bridged . . . 49
4.10.3 WiFi . . . 51
4.10.4 Summary . . . 52
4.11 Distributed Mode . . . 52
4.11.1 VM Scheduling . . . 53
4.11.2 Communication . . . 53
4.11.3 ZeroMQ Communication Patterns . . . 53
4.11.4 Distributed Architecture . . . 54

5 Implementation **57**
5.1 Implemented Architecture . . . 57
5.2 Scenario Config . . . 60
5.2.1 Example Scenario Config . . . 60
5.2.2 Scenario Config Binding & API . . . 62
5.3 Node System . . . 62
5.3.1 Qemu . . . 65
5.4 NetworkConfigurators . . . 66
5.5 NetworkBackends . . . 67
5.5.1 VDE . . . 67
5.5.2 Bridged . . . 69
5.5.3 WiFi . . . 82
5.6 Distributed . . . 85
5.6.1 State Register . . . 86
5.6.2 State Echange . . . 86
5.6.3 State Start Nodes . . . 86
5.6.4 State Distance Matrix . . . 87
5.6.5 Distance Matrix Mapping . . . 88
5.7 VM Preparation . . . 88

5.8 CLI 90
5.8.1 Event System 90
5.8.2 CLI & Error Checking 91
5.8.3 MiniWorld API Demonstration 94

6 Evaluation **97**
6.1 Test Environment 97
6.2 QEMU 98
6.2.1 Image Comparison 100
6.2.2 Snapshot Boot 101
6.3 QEMU NIC Models Bandwidth 102
6.4 Network Backends 103
6.4.1 Bandwidth 104
6.4.2 Round Trip Times 105
6.4.3 Distributed mac80211_hwsim showcase 106
6.4.4 Connection Switching Times 109
6.4.5 Bridged WiFi Network Backend 110
6.4.6 Bridged LAN Network Backend 111
6.4.7 Summary 112
6.5 Distributed Emulation 112
6.5.1 Distributed Coordination 113
6.5.2 Test System 2 114
6.5.3 Boot Times 114
6.5.4 Network Switching 115
6.5.5 Tunnel Delays 116
6.5.6 Serval Study 117
6.6 Summary & Best Practice 119
6.6.1 Boot Mode 119
6.6.2 Network Backend 120
6.6.3 Distributed Mode 121

7 Conclusion **123**
7.1 Future Work 123

Appendix **125**

Bibliography **129**

Eidesstattliche Erklärung **133**

List of Figures

2.1 Model for Network Emulators [28] 5
2.2 Electromagnetic Wave Spectrum [41] 7
2.3 Radio Propagation [16] 8
2.4 802 Family [17] 8
2.5 Hidden and Exposed Terminal Problem [41] 10
2.6 802.11s Power Management Scenario [19] 12
2.7 Linux Network Internals 13
2.8 Integration Of A Driver In The Linux Kernel Network Stack [44] 15
2.9 Linux Wireless Design [26] 15
2.10 Type 1 Hypervisor (left side) vs. Type 2 Hypervisor (right side) [40] 17
2.11 Full Virtualization vs. Paravirtualization [40] 18

3.1 VDE Architecture [11] 23
3.2 Open vSwitch Architecture [34] 24
3.3 WiFi Virtualization Approaches [48] 25
3.4 Virtual WiFi Architecture [48] 25
3.5 Wmediumd architecture 26
3.6 CORE EMANE Integration [3] 29
3.7 Ns-3 Tap Bridge Model [42] 30
3.8 Netkit Architecture [35] 32
3.9 NEMAN Architecture [36] 33

4.1 MiniWorld Architecture 36
4.2 MiniWorld Workflow 37
4.3 Simulation Lifetime Sequence Diagram 40
4.4 Step Sequence Diagram 42
4.5 Sequence REPLable 44
4.6 VDE Network Backend 48
4.7 Bridged Network Backends 49
4.8 WiFi Network Backend 52
4.9 MiniWorld Architecture Distributed 54

5.1 MiniWorld Architecture Implementation 58
5.2 Config System 62
5.3 Nodes 63
5.4 NetworkBackendBootStrapper Bridged 70
5.5 Bridged Network Backend Class Diagram 71
5.6 Bridged Network Backend Switch And Connection Class Diagram 72
5.7 Ebtables & TC Interaction 80
5.8 Node 2 (2 Connections) TC Setup 81
5.9 WiFi Network Backend Interaction 83
5.10 Event System Interaction 92

6.1 Boot Mode Times OpenWrt Barrier Braker, 32MiB RAM 99
6.2 Boot Modes CPU, OpenWrt Barrier Braker, 32MiB RAM 100
6.3 OpenWrt vs. Debian 8 Boot Times. OpenWrt RAM: 32MiB. Debian RAM: 256MiB. Boot Mode: *Selectors Shell Prompt* 101
6.4 Boot Times CPU Usage (+RD): OpenWrt Barrier Braker vs. Debian 8 101
6.5 Boot Modes Debian 102
6.6 Boot Modes Debian CPU 102
6.7 Qemu NIC Models Bandwidth 103
6.8 Network Backend Throughput 104
6.9 Network Backend Delays 105
6.10 Wireshark Capture Of 802.11s Beacon Frames and ICMP Ping 107
6.11 Distributed mac80211_hwsim Showcase 108
6.12 Differential Switching 109
6.13 Network Switching Topologies (WiFi Network Backend) 110
6.14 Network Switching Modes Single 111
6.15 Network Switching Modes Multi 112
6.16 Distributed Coordination Step Times 113
6.17 Distributed Boot Times, 40 OpenWrt BB Nodes, RamDisk, 128MB RAM, Selectors Boot Prompt, NodeDistributionEqual, Test System 2 115
6.18 Distributed Differential Network Switching, 40 OpenWRT BB Nodes, Test System 2, Fixed-Range Model, Bridged WiFi Network Backend, No Link Quality Impairment, 2 Runs Only 115
6.19 Distributed Mode: Tunnel Overhead, 5 OpenWRT BB Nodes, Test System 2, Fixed-Range Model, Bridged WiFi Network Backend, No Link Quality Impairment 116
6.20 Serval Peer Discovery, Distributed Chain 256, 7 Computers 117
6.21 CPU Usage Serval Chain 256 118
6.22 CPU Usage Serval Chain 512 119

List of Listings

1 B.A.T.M.A.N. Advanced Scenario Config 62
2 EmulationNode Start 64
3 Management Network Interface Setup By *EmulationNodeNetworkBackend* . . 64
4 Qemu Overlay Image Command 65
5 Qemu And VDE Commands (Node 1 only) 67
6 VDE Network Setup 68
7 VDE REPL Commands Output 69
8 ShellCommandSerializer Chain 2 74
9 Ip Batch Mode 74
10 Tc Batch Mode 74
11 ShellCommandSerializer Example 76
12 QemuTap Start (Node 2) 77
13 Bridged LAN Network Backend Chain 3 Setup 78
14 Bridged WiFi Network Backend Chain 3 Setup 79
15 Linux Ebtables and TC Commands Link Quality Model WiFi 81
16 Ebtables and TC Commands: Link Quality Model WiFi 82
17 Frame Format 84
18 Netlink Callback 85
19 ZeroMQ Server 86
20 ZeroMQ Server Router Distance Matrix 87
21 ZeroMQ Server Publish Distance Matrix 88
22 Distance Matrix To Node Mapping 88
23 Image Deployment 89
24 Grub Modifications at /etc/default/grub 89
25 Systemd Modifications 90
26 Sysvinit Modifications 90
27 Event System Demo 91
28 CLI Action Method 92
29 CLI Action Class 93
30 CLI Action Decorator 94
31 CLI Progress View 94
32 Use Case 95

33 Get Qemu NIC Models 103
34 802.11s Peer Link Blocking 106
35 WiFi Network Backend 802.11s Scenario Config 107
36 802.11s Mpath Dump 108
37 Setup Routing For Node 1 To Node 3 Via 2 116

38 Scenario Config Template 125

1 Introduction

Simulation[1] is the ”Imitation of a situation or process” and ”The production of a computer model of something, especially for the purpose of study”. Emulation[2] in contrast, is defined as the ”Reproduction of the function or action of a different computer, software system, etc.”.
Emulation is used in computer science to mimic the behaviour of hardware/software which is not supported any more by modern technologies. The approach is known from terminal emulation or video game platforms emulators such as for the Super Nintendo or the old fashioned Game Boy.
Network emulation is defined by Lochin et al. as the emulation of the network layer and above while the lower layers are simulated [28]. *Network simulation* in contrast leverages a model to embrace the core of the studied problem. While network simulation is good in the early design process of new protocols, it can not replace network emulation since real implementations need to be tested. The FreeBSD TCP stack has been verified by network simulation but nevertheless a bug has been found which showed up under certain network conditions[9]. Hence, testing the actual implementation is crucial.
Network simulation and emulation are cost-efficient since either a cheap model replaces the need of hardware, or the hardware is emulated. Both approaches allow testing of network protocols, routing algorithms and application software under certain network conditions. Simulating or emulating wireless communication requires additionally mobility to be incorporated.
In this thesis, a network distributed network emulation framework called **MiniWorld** is presented. The main contributions can be summarized as following:

- KVM based virtualization enables nearly every software and hardware to be emulated.
- 4 network backends prove the flexibility, modularity and exchangeability of Mini-World's design.
- Both wired and wireless communication is supported.
- Mobility patterns, link impairment and especially WiFi virtualization, ease the evaluation and development of Wireless Mesh Networks.
- The emulation can be distributed across several computers with an emulation aware scheduling algorithm.
- Connection tracking, address configuration and network supervision features are provided automatically to every network backend.

The remainder of this thesis is organized as follows: First the basics of network simulation, emulation physical testbeds, wireless technologies and networking under Linux are outlined in Section 2. Related work in the field of network simulation and emulation is introduced in Section 3. The workflow and design of MiniWorld is presented in Section 4. Implementation details are discussed in Section 5. Finally, MiniWorld is experimentally evaluated in Section 6 and a conclusion as well as future work is outlined in Section 7.

[1]`https://en.oxforddictionaries.com/definition/simulation`. Last viewed on 07.12.2016
[2]`https://en.oxforddictionaries.com/definition/emulation`. Last viewed on 07.12.2016

2 Basics

This section introduces basics which are necessary to understand this thesis. First layered network architectures are introduced briefly in Section 2.1. They establish the basis for the understanding of network communication which is required to describe network emulation. Network simulation, emulation and testbeds are discussed in Section 2.2 where MiniWorld is classified according to a model for network emulators.
Since MiniWorld aims to be a wireless emulator, the principles of wireless communications are introduced in Section 2.3. The Section discusses radio propagation effects and introduces standards such as *802.11* and *802.11s*.
Since MiniWorld leverages functionality of the Linux kernel for some of its network backends, the fundamentals of Linux bridges, the flow of packets through the *netfilter* framework, Traffic Control (TC) and the integration of wireless drivers into the Kernel are depicted in Section 2.4.
Node virtualization technologies such as container and hypervisor-based virtualization are introduced in Section 2.5 to the reader.

2.1 Layered Network Architectures

The use of layers in computer networks is common. They are used to reduce overall complexity. Each layer offers a certain service. In the following a short summary of the OSI protocol stack is given because the layer nomenclature is required to describe network communication.
Hosts communicate with a protocol. It is an agreement on communication rules. Actually, while hosts communicate, a whole protocol stack is used. Higher layers pass their Protocol Data Unit[1] down towards lower layers. While the PDUs traverses the protocol stack, each layer adds a new encapsulation to the PDU. Finally it is sent via the *Physical Medium*. On the receiving host, the layers of the corresponding protocols are removed one after another. The standard model is the Open Systems Interconnection (OSI) reference model and contains seven layers [41]. The lowest layer (layer 1) is the *physical layer*. It converts binary data according to the used communication channel. For example for wireless networks, bytes are modulated and sent via electromagnetic waves.
The next higher layer (layer 2) is the *link layer*. It is mostly responsible for managing the channel coordination. For that purpose it includes a sublayer called Medium Access Control (MAC). Layer 3, the *network layer* is responsible for routing packets. Routing solves the problem of how packets can be transmitted to a specific destination.
The *transport layer* (layer 4) is responsible for the actual data transport. The layer handles *network congestion*, *flow control* and *reliability*.
Layers 5 and 6 in the OSI model are called *session* and *presentation* respectively. The first deals with session management while the latter is concerned with the presentation of data.

[1]The Protocol Data Unit (PDU) of layer 1,2,3 and 4 are called *bits/symbols*, *frames*, *packets* and on layer 4 *segments* for TCP and *datagrams* for UDP

On the last layer (7), the communication processes agree on a protocol such as HyperText Transfer Protocol (HTTP).[41].

2.2 Simulation vs. Testbeds vs. Emulation

According to Lochin et al. there are three possibilities to carry out network experiments: Simulation, emulation and testbeds lochin2012should. The approaches are explained in the following in more depth.

2.2.1 Network Simulators

Network simulators rely on a virtual clock. For that reason they are also called *discrete-event simulators*. They are not tied to the clock of the system, hence dependent on the complexity of the problem that shall be simulated, time can be simulated in discrete steps either faster or slower than system time. For example a work-day model[2] could be simulated in a few seconds or in a few days.
A simulator is cheap and scalable. It simplifies the studied problem by trying to embrace the core of the problem. Applications have to be rewritten to match the API and to leverage the network models of the simulator. This approach allows for development of prototypes, but cannot replace testing the actual code due to the fact that the overhead in real system hardware needs to be evaluated as well as bugs have to be found [28]. Remember the example given in the introduction where the FreeBSD TCP Stack has been evaluated inside a simulator but a bug only appeared in the emulation, not the simulation approach.
Examples of network simulators are summarized in Chapter 3.1.

2.2.2 Testbeds

The simplest approach to perform a network experiment is to reuse existing infrastructure. This allows testing real code and realistic network models. For example a mesh network infrastructure could be built to evaluate some mesh software. This results in a realistic model of layers 1 and 2 but has many disadvantages such as the cost and the inflexibility. Creating a certain network topology requires much effort and scenario changes based on certain events like time is not possible. Furthermore, the integration of movement models into this approach is very hard.
The paper "When should I use network emulation?" [28] points out a situation where a new technology is still in development while applications shall be tested on that particular platform. This is impossible with testbeds but a simulator or emulator can be used instead.

2.2.3 Network Emulators

Emulation combines the best of simulation and testbeds: The actual application code can be tested in network emulators while the physical infrastructure is used by simulating only parts of the network.
In contrast to simulators, network emulators do not have a virtual clock. While simulators use a simulation engine to provide a model which allows testing only inside a closed world, network emulators rely on the functionality of the Operating System (OS) . Therefore, real

[2]Tries to match the behaviour of people's work day by means of movements.

protocol code can be evaluated without any modifications of the binaries.
Typically the nodes in the emulated network are provided by means of virtualization. This enables multiple nodes to co-exist on a single computer. A network model is achieved through *artificial impairments* such as bandwidth reduction, delay, jitter and/or packet loss [28].
Network emulation is younger than network simulation. Computer resources were limited and posed higher limits on virtualization.
Lochin et al. define network emulation as the reuse of layers 3 and above while simulating layers 1 and 2 [28]. Network simulators in contrast simulate all layers. Lochin et al. propose a general model for network emulators which is introduced in the following to the reader (Section 2.2.3). Finally MiniWorld is classified according to the model (Section 2.2.3).

Network Emulation Architecture Model

The architectural model proposed by Lochin et al. [28] is depicted in Figure 2.1. The upper part of the Figure shows the *System under Test* which is not part of their model.
There are three important components in their architectural model: The *Hardware Model*, the *Impairments Model* and the *Network Model*. These components are described bottom-up in the following.
Hardware Model: The *Hardware Model* describes the actual hardware. Virtual resources such as virtual nodes and links are mapped to the physical hardware. They distinguish between a centralized and distributed approach and argue that distribution offers the better mapping of virtual resources but also introduces more complexity. Beneath the fact that distributed systems are always harder to debug, there is also the problem of clock synchronization.
Impairments Model: There are 2 impairment models which can be applied in either the kernel- or user-space of the OS.
The first is called *Artificial QoS* and more a way of unit testing. An example given is an induced packet loss for testing TCP acknowledgements. The second is *Realistic QoS* which aims at imitating the characteristics of a special network (wired, wireless, satellite network etc.) as accurately as possible.

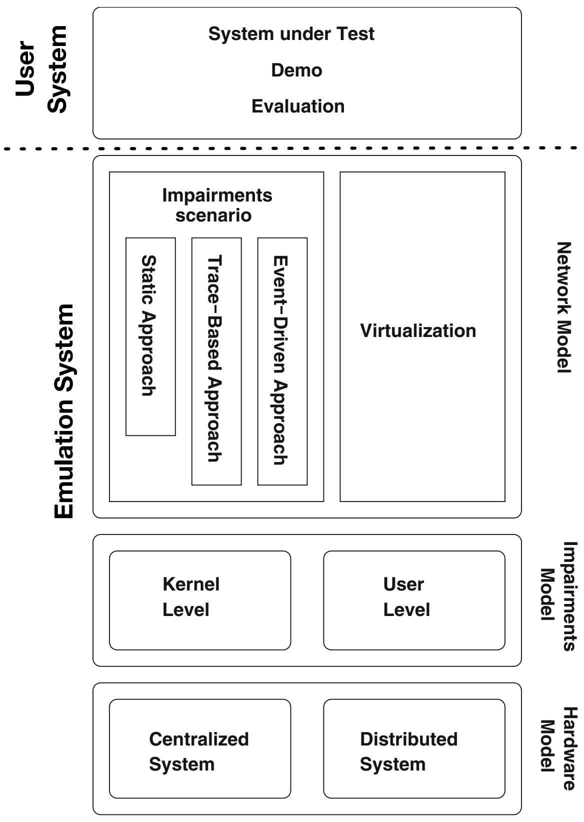


Figure 2.1: Model for Network Emulators [28]

Moreover, they argue that kernel-space impairment has the advantage of being more accurate due to the timer resolution of the Linux kernel.
Network Model: The *Network Model* is further divided into *Virtualization* and *Impairments scenario*. The virtualization technology is used to simulate virtual hosts and links. There are

three kinds of impairment approaches: *static*, *event-driven* and *trace-based*. The first consists of parameters such as bandwidth and delay which remain constant during the experiment. The second changes the impairment based on events, e.g. every second. The *trace-based* impairment captures the characteristics of a certain network and applies the observed behaviour on the packets during the emulation. Moreover, event-driven and trace-based impairment can also be used together [28].

MiniWorld Classification

Lochin et al. argue that one should use simulation for modeling and prototyping and emulation for the evaluation of the real application/protocol implementation [28].
The paper points out an architectural model of network emulators by which MiniWorld can be classified as follows: It can be used as a centralized system with only a single computer as well as a distributed system. There are 4 network backends included: The **Virtual Distributed Ethernet (VDE)** network backend provides a virtual switch which performs impairment in the user-level. The **Bridged** network backends however leverage kernel-level impairment. The **WiFi** network backend does not have an impairment at the time of writing. Both the VDE and *Bridged* network backends use an event-driven approach. The virtualization technology used by MiniWorld is Kernel-based Virtual Machine (KVM).

2.3 Wireless Technologies

The following Section provides insights into wireless technologies since MiniWorld aims at being a wireless emulator. First, the concept of electromagnetic waves are introduced to the reader. Afterwards radio propagation effects are illustrated. The following Section is based on the book ”Computer Networks” [41] from Tanenbaum et al.

The basics of wireless communication are waves that propagate through space. When electron moves, they create electromagnetic waves. This effects was predicted by Maxwell in 1865 and first observed by Hertz in 1887. A wave has a frequency which describes the oscillations per second. The unit for the frequency is *Hz*. Furthermore, waves have different *wave lengths* which is the difference between 2 consecutive maxima. The relation between frequency and wave length is defined by $frequency * wave\ length = speed\ of\ light$. WiFi on 2.4GHz therefore has a wave length of approximately 0.125 meters whereas 5GHz wireless networks have a wave length of approximately only 0.06 meters.
The bandwidth of a a frequency spectrum is the difference between the highest and the lowest frequency.
Shannon describes the capacity of a noisy channel as follows: $capacity = B * log_2(1 + S/N)$, where S/N is the so called Signal-to-Noise Ratio (SNR) and B the bandwidth. Therefore, the amount of information a signal can carry is better if there is more bandwidth and signal strength and as little as possible noise. Figure 2.2 outlines the spectrum of electromagnetic waves. Frequency is depicted in a logarithmic unit on the top part of the Figure. Frequencies such as (from left to right) *Radio*, *Microwave*, *Infrared* and *Visible light* can all be used for transmitting information. *UV*, *X-ray* and *Gamma ray* are dangerous to human livings and do not propagate through buildings very well but have an even higher bandwidth.
The bottom part of the Figure shows the technologies used inside the specific ITU frequency spectrum. *Fiber optics* are very fast because they provide a high bandwidth (lower right part of the Figure). WiFi belongs to the *UHF* and *SHF* band which have frequencies between

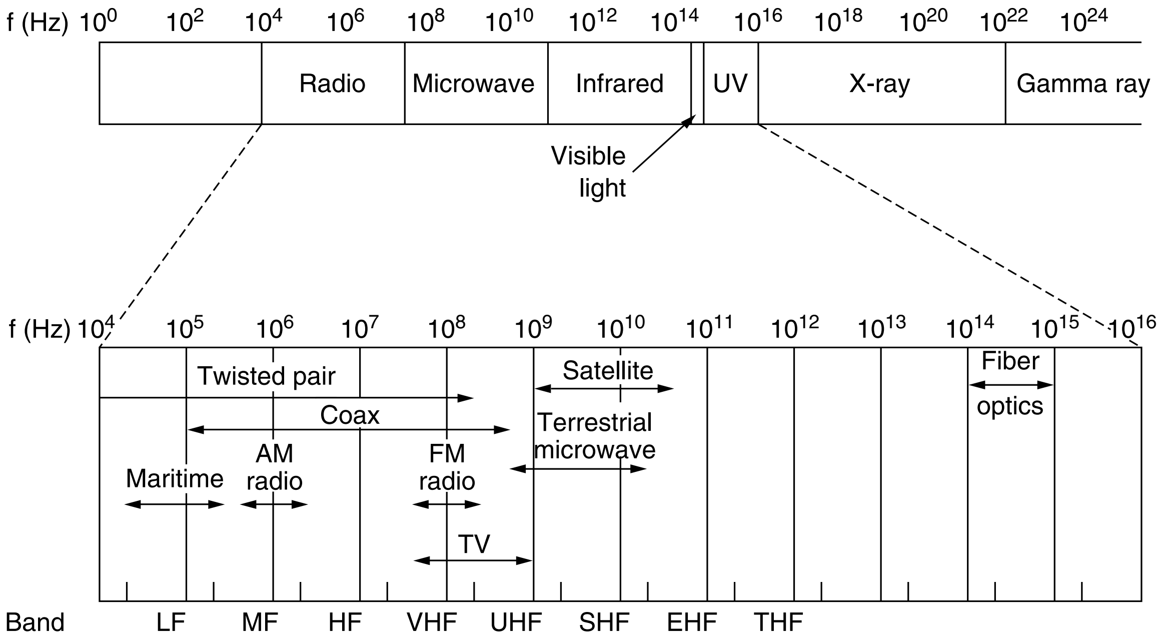

f (Hz) 10^0 10^2 10^4 10^6 10^8 10^{10} 10^{12} 10^{14} 10^{16} 10^{18} 10^{20} 10^{22} 10^{24}
Radio
Microwave
Infrared
UV
X-ray
Gamma ray
Visible light
f (Hz) 10^4 10^5 10^6 10^7 10^8 10^9 10^{10} 10^{11} 10^{12} 10^{13} 10^{14} 10^{15} 10^{16}
Twisted pair
Coax
Satellite
Terrestrial microwave
Fiber optics
Maritime
AM radio
FM radio
TV
Band
LF
MF
HF
VHF
UHF
SHF
EHF
THF

Figure 2.2: Electromagnetic Wave Spectrum [41]

300MHz-3GHz (UHF) and 3GHz-30GHz (SHF) [41].
Nowadays WiFi operates mostly in 2.4GHz or 5GHz frequencies. The bands are also called *ISM* because they are license-free and intended for **I**ndustrial, **S**cientific and **M**edical purposes [17].
The properties of radio waves (3KHz-300GHz) depend on the frequency: With low frequencies they can pass through obstacles well while this reduces with higher frequencies and lets them bounce off obstacles. The reduction of signal strength through a medium is called *path loss* [41]. According to Vijay et al. there are three attributes which influence the propagation of radio waves and are depicted in Figure 2.3: *reflection*, *diffraction* and *scattering*. These phenomena are part of our everyday lives. Light is *reflected* by mirrors (smooth surface) (a). *Scattering* (c) is the reason why the sky is blue. Short waves (blue and violet light) are scattered by nitrogen and oxygen in the atmosphere. Scattering means the bouncing of a wave in all directions. Diffraction (b) is the spreading and bending of a wave around an object. Moreover waves may also be refracted. This phenomenon occurs when a wave changes its medium since the propagation speed changes. An object placed only half in water shows the effect where the wave changes the angle in the other medium. Finally, waves may also be absorbed [32].
The propagation of waves therefore does not occur solely in one direction. Waves may arrive delayed and out of phase and cancel the actual signal. This effect is called *multipath fading* [41].
Creating propagation models for wireless communication is very complex and depends on the environment. For example the propagation is different in free space compared to inside a building.

The actual transmission of data via electromagnetic waves is achieved by *digital modulation* which converts binary data into an analog signal.

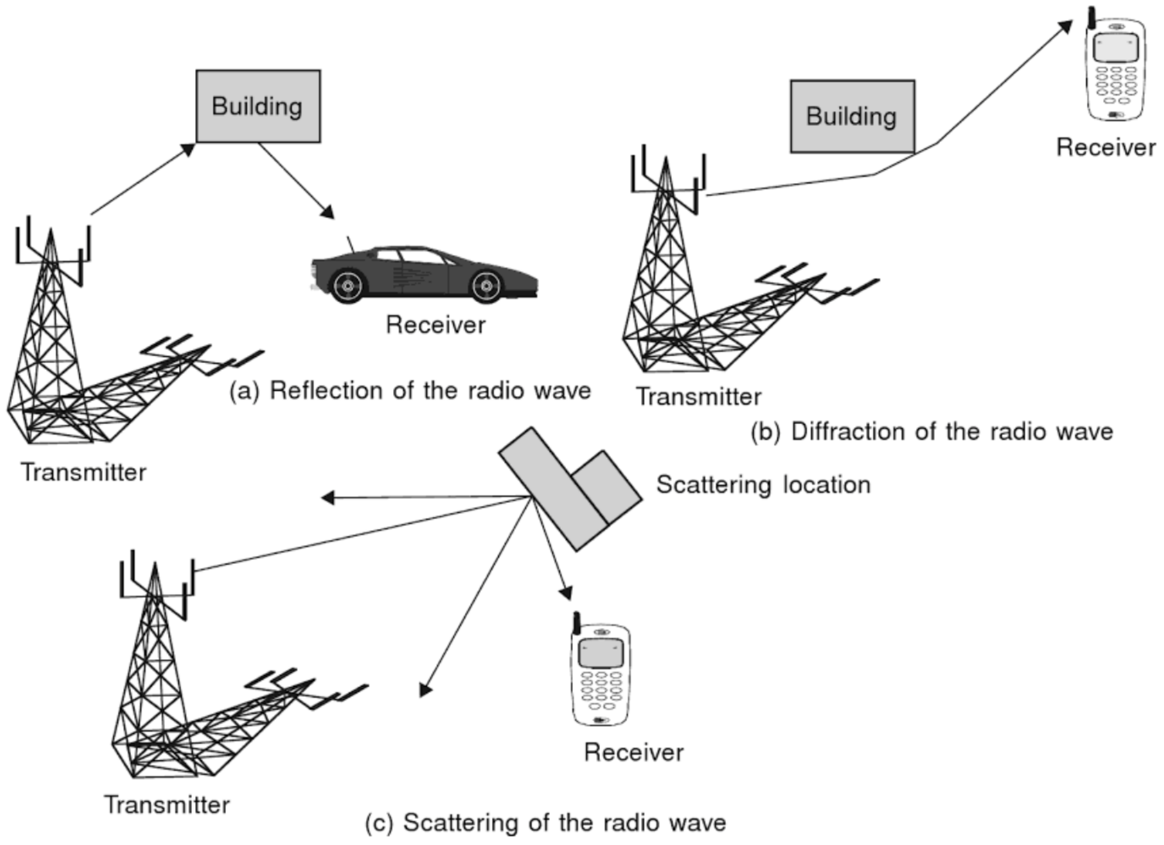

Building
Receiver
(a) Reflection of the radio wave
Transmitter
Building
Receiver
Transmitter
(b) Diffraction of the radio wave
Scattering location
Transmitter
Receiver
(c) Scattering of the radio wave

Figure 2.3: Radio Propagation [16]

2.3.1 IEEE 802

The IEEE 802 family defines common physical and link layer components such as *Ethernet* (802.3) whose MAC is known as *CSMA/CD*. *802.5 is Token Ring* and *802.11* WiFi. Figure

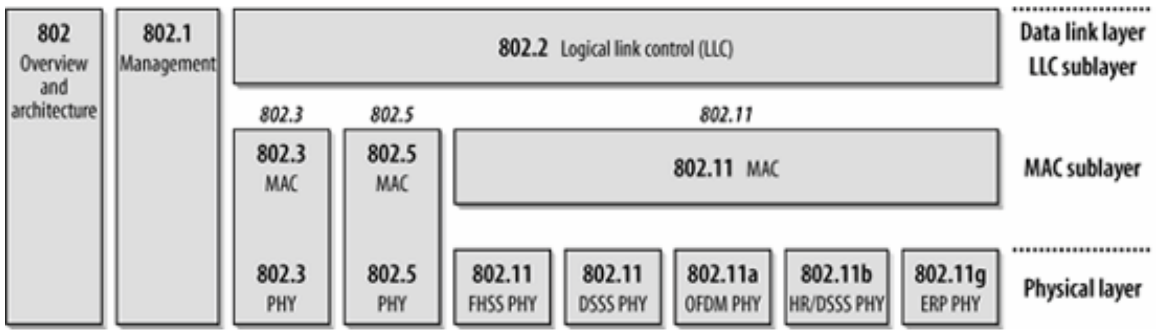


Figure 2.4: 802 Family [17]

2.4 shows that there are many physical layers (lower right part of the Figure) available for wireless devices. These differ in bit rates, frequencies and the used modulation schemes. The Figure shows layer1 and 2 of the OSI model. Note that layer 2 is further divided into the MAC and Logical Link Control (LLC) layer. The MAC layer coordinates the access of the medium [17] and the LLC layer primarily identifies the upper layer protocol but was designed to hide the differences of the lower layers.

In the following, first the *802.11* standard is depicted. Afterwards the IEEE amendment *802.11s* for mesh networks is introduced to the reader.

802.11

The IEEE standard for wireless networks is 802.11. Well known extensions are e.g. 802.11/a-/b/g/n/ac.
In the following first the modes of operation are illustrated. Afterwards the MAC layer is explained in depth.

Operation Modes

The first operation mode is the well known pattern where stations talk to a central, mostly static **Access Point (AP)**. The second mode is known as **Ad Hoc mode** where no fixed infrastructure is needed. Instead, stations in range can talk directly with each other [41]. If stations are out of signal range but in range of another station, additional routing protocols like Better Approach To Mobile Adhoc Networking (B.A.T.M.A.N.) or Optimized Link State Routing Protocol (OLSR) enable the creation of multi hop networks known as Mobile Ad Hoc Networks (MANETs). **WiFi-Direct** is an operation mode in which one of the clients that want to communicate with each other, creates a software AP. A special **Mesh** mode is based on the *802.11s* standard. With the *Mesh* mode a new kind of networks called Wireless Mesh Networks (WMNs) arose. MANETs and WMNs are very similar but there are also some differences. Basically, WMNs in addition to the MANETs, include some kind of fixed routers with no energy constraints. They are used to connect the mobile nodes to a wired network. This enables offloading from mobile stations with limited battery for services which require more resources [30].

MAC

Wireless in contrast to wired stations are half duplex. The reason for this is that a wireless Network Interface Card (NIC) cannot transmit and receive simultaneously because the antenna size correlates with the wave length. Therefore, in contrast to Carries Sense Multiple Access/Collision Detection (CSMA/CD) where collisions can be detected, 802.11's MAC relies on *channel sensing*. The mechanism is called Carries Sense Multiple Access/Collision Avoidance (CSMA/CA).
In CSMA/CA, each station senses the wireless channel for a short time (DCF InterFrame Spacing (DIFS)). If the channel is free, the station can send frames. To prevent stations from sending simultaneously, 802.11 introduced an early backoff mechanism. Collisions are expensive because always a whole frame is transmitted since collisions cannot be detected. If a collision occurs, a station performs exponential backoff. After a frame has been sent, it is acknowledged by the receiver to indicate a successful transmission and all other stations that the transmission is over. The mechanism is known as Distributed Coordination Function (DCF) due to the fact that no central coordinator is needed. Figure 2.5 shows 2 problems in wireless networks. The first is called *hidden terminal problem* because stations may be out of receiving range from each other but overlap at a station B in the case of the Figure. The problem is that although C is transmitting to B, A starts a transmission to B as well because it cannot sense the transmission of C.
The second problem is called *exposed terminal problem*. It can arise when an overwhelmed

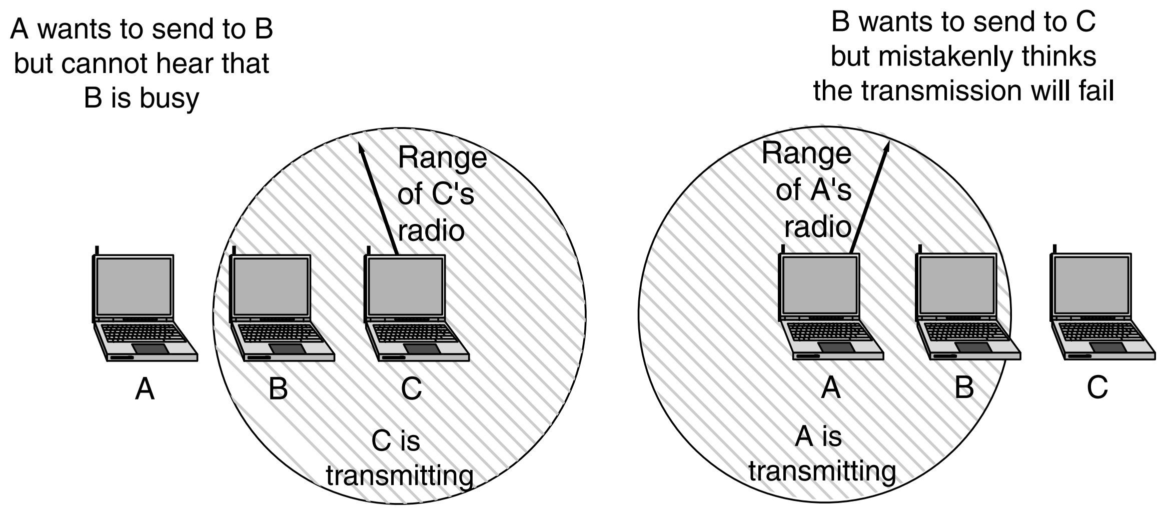


Figure 2.5: Hidden and Exposed Terminal Problem [41]

station may misleadingly decide that a station is busy. The scenario is depicted in the right side of Figure 2.5. A is transmitting to D (not shown). Therefore B, which wants to send to C, decides that the channel is busy. The problem is the ambiguity about which station is sending and results in a missed opportunity to send a frame.
These kinds of ambiguities are reduced by letting each station sense physically and virtually. The first is the sensing on the channel itself for a valid signal while the latter is achieved by listening to frames which stations in range sent. All frames include a Network Allocation Vector (NAV) field which indicates how long the transmission will take. This approach shall eliminate the *exposed terminal problem*.
An optional RTS/CTS mechanism tries to reduce the *hidden terminal problem* by means of adding messages which check if the channel is free. The idea behind the approach is to send small frames, where a collision can be neglected. The Ready To Send (RTS) message indicates that a sender wants to start a transmission. The AP controls the channel and sends the station a Clear To Send (CTS) message. All stations in range of the AP hence know that the channel is busy. Note that this mechanism is not often used in practice because it does not help for short frames and may slow down additionally. Moreover, CSMA/CA already slows down senders by means of exponential backoff in case of a collision.
There are 2 more mechanisms to improve the overall transmission throughput. The first is *rate adaption*. There are different modulation schemes used for different SNR quotients which vary in terms of throughput and the likelihood that a transmission may fail for a given SNR. For example 802.11a has rates ranging from 6 to 54 Mbps.
To prevent slow senders from slowing down a fast sender, there is a mechanism called transmission opportunity (TXOP) which allocates each station the same amount of channel time also known as *air time*. Tanenbaum et al. outline an example where a sender with 6Mbps slows down a sender with 54Mbps because both are allowed to send the equal number of frames. Since the slow sender is nine times slower both end up with a speed of 5.4Mbps ($54 * 1/10 = 6 * 9/10$) while with TXOP enabled they end up with a speed of 3Mbps (6Mbps/2) and 27Mbps (54Mbps/2). The second improvement is the use of *fragmentation*. The likelihood for small frames to result in a successful transmission is higher due to the noisy channel. Therefore frames are split into fragments whereby each has an own checksum and sequence number so that frames can be acknowledged via a stop-and-

wait protocol.
Mobile nodes may run on battery and hence energy consumption plays an important role. Therefore energy management is built into 802.11. APs periodically advertise their presence in form of a *beacon frame* which includes the Service Set Identifier (SSID) in each beacon frame as well as information about the encryption, when the next beacon frame is sent and additionally a traffic map. This map shows each station whether the AP has buffered frames while the station has been in sleep mode. The station can then fetch the frames via a poll message from the AP. A sleep mode can be entered by setting a flag in a frame destined to the AP. Due to the beacon frame, the stations knows how long it can stop listening to the channel.
There is a special mode called Automatic Power Save Delivery (APSD) where the AP lets a station know of the buffered frames by sending a frame directly after it received one from the station. This allows for frequent traffic such as VoIP to fetch buffered frames faster than the beacon interval is.
Finally there is Quality of Service (QoS) built into the 802.11 protocol. Depending on the priority there are frame spaces which are shorter or longer than the DIFS frame spacing. *Interframe spacing* gives high priority traffic such as VoIP the opportunity to send before others by waiting only until the specific time instead of the default DIFS [41].

802.11s

Mesh networks are very different compared to AP-based networks. Every node is assumed to take part in the process of forwarding frames so that a multi hop network can be created. In 2011, IEEE published an amendment[3] for mesh networks called *802.11s*. Unfortunately, the author did not have access to the official document, therefore a whitepaper from Henry is used as source in the following.
In contrast to IP-based routing, *802.11s* focuses on the link layer for routing, called *Path Selection* because it operates on layer 2. The mandatory Hybrid Wireless Mesh Protocol (HWMP) consists of both proactive and reactive components. In the first approach, a route is discovered on demand only in contrast to the proactive approach [19]. HWMP is based on Ad Hoc On Demand Distance Vector (AODV) [20].
Peer discovery is achieved by beacon frames, where mesh stations advertise their presence as well as mesh specific settings. Afterwards a peering between the mesh stations is done with special peering frames. The peering contains information about the supported rates, power management capabilities, supported channels, mesh security etc.
802.11s comes with a new authentication scheme called Simultaneous Authentication of Equals (SAE) which is based on the *Dragonfly* authenticator and uses a shared secret to secure the communication between 2 peers. Moreover, 802.1X can be used instead but requires stations to be in range of the authenticator service.
Furthermore, channel access coordination is done by a new Distributed Coordination Function called Mesh Controlled Channel Access (MCCA).In a Wireless Mesh Network some nodes may be running on battery such as cell phones or sensor nodes. Therefore, extra energy modes have been developed. Figure 2.6 shows a mesh network with three different energy modes. In the *Active mode* no power management is done, hence the station participates in frame forwarding and path discovery. The *Light sleep mode* is comparative to the APSD mode in the standard 802.11 protocol: The mesh station can sleep during beacon

[3] `http://standards.ieee.org/findstds/standard/802.11s-2011.html`

frames and then fetch the buffered data from its neighbours. In a more advanced sleep mode, a node can decide to not take part in the frame forwarding process. It only sends and receives frames destined for itself.
Figure 2.6 points out that mesh stations may signal neighbours individual power manage-

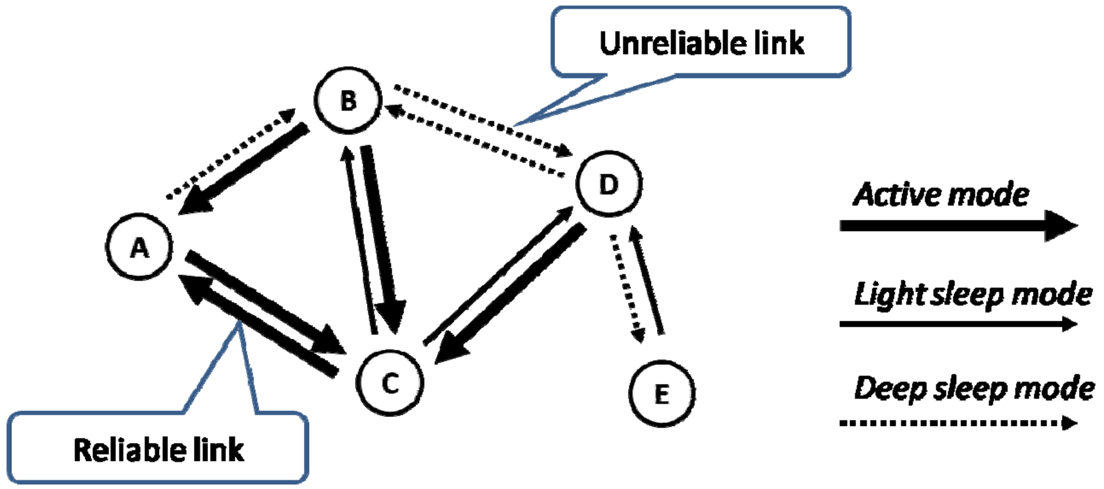


Figure 2.6: 802.11s Power Management Scenario [19]

ment settings. The link between B and D is unreliable, therefore both enter the *Deep sleep mode* for each other. C enters *Light sleep mode* for both B and D.
Routing therefore incorporates energy constraints of mobile nodes. HWMP in contrast to IP routing, integrates not only the hop count but also the so called *airtime metric*. The airtime incorporates data and bit error rate between each pear. In the default reactive mode, a route is only determined fon demand. If a station A wants to send to a station Z, a Path Request (PREQ) is broadcasted to each peer to determine the path to Z. All nodes which receive the PREQ forward the request until it reaches Z. At the end, multiple PREQs may arrive at Z, but a Path Reply (PREP) is only send back through the best path. The past path is determined by means of hop count and metric. The metric for a given link is added by each mesh station which forwarded a path selection frame. The *airtime metric* makes 802.11s special because the wireless nature is integrated into the routing algorithm [19].
To provide a short conclusion, 802.11s is the IEEE amended standard for Wireless Mesh Network. A first implementation[4] is already integrated into the Linux kernel [20]. 802.11s comes with a routing algorithm that takes the wireless nature into account for routing decisions. This enables 802.11s to use a different network layer protocol than IP, because it uses the mac layer for routing. Moreover new power management nodes tailored towards the special energy constraints in Wireless Mesh Networks have been developed.

2.4 Networking in Linux

The functionality of Linux is used to implement the emulation properties of MiniWorld. The basics which are required to understand the implementation details of MiniWorld are outlined in the following. First, ethernet bridges and the flow of frames through the Linux kernel are introduced in Section 2.4.1 to the reader since Linux bridges are used by both

[4]`http://www.o11s.org`

Bridged network backends.
Linux's TC is used as link impairment in the *Bridged* network backends. A brief introduction to the TC system is provided in Section 2.4.2. The **WiFi** network backend relies on a Linux WiFi driver, hence the driver architecture of Linux is illustrated in Section 2.4.3.

2.4.1 Bridge Netfilter Hooks

Ethernet bridging (802.1d) allows LAN segments to be interconnected on the data link layer. A bridge is the implementation of a software switch inside Linux. In Linux, a bridge is represented as just another NIC. The user-space commands *brctl* and *ip* (iproute2 package) can be used to enslave a NIC to the bridge and control the internal bridge options.
An interface can only be enslaved to one bridge.
Netfilter is a Linux framework which offers kernel modules several hooks: By registering callback functions, they are integrated into the network handling code.
Figure 2.7 shows netfilter hooks used for the *ebtables* and *tc* commands. The first enables

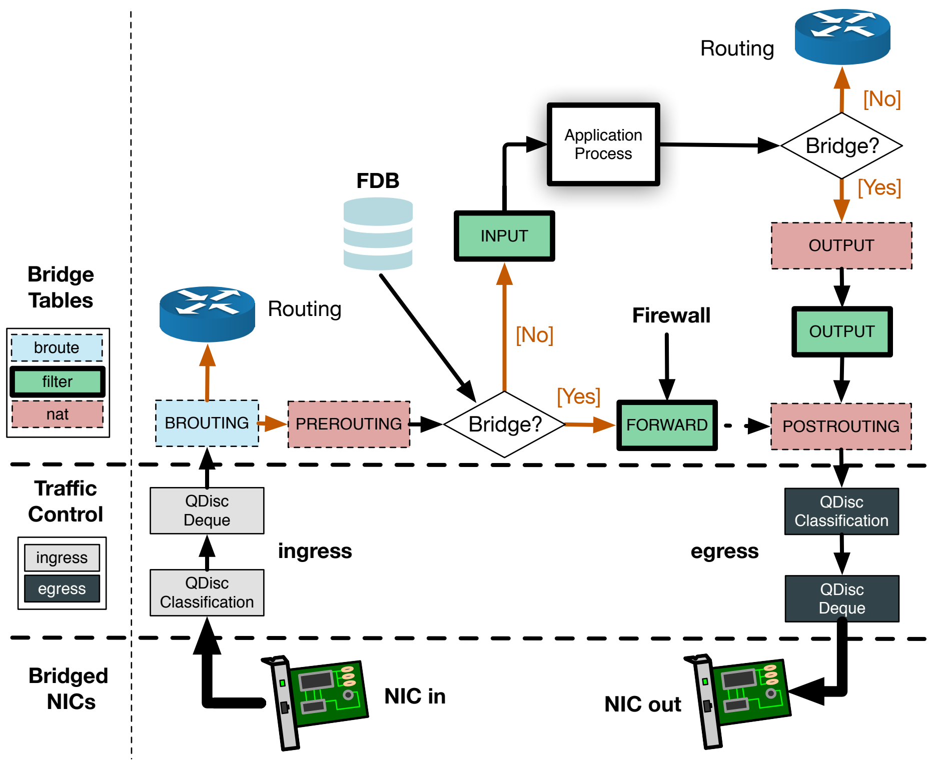


Figure 2.7: Linux Network Internals

the administration of a link layer firewall while the latter offers traffic control such as bandwidth limiting and QoS.
Moreover, the Figure outlines how packets traverse parts of the Linux network stack. There are three different levels depicted in the Figure. On the lowest level packets are sent and received from a NIC. The next layer shows the integration of the Linux traffic control facilities. First frames/packets are classified and afterwards a Queuing Discipline (QDisc) algorithm schedules the packets.
The upper layer shows the hooks used by *ebtables*. There are three different tables: *broute*, *filter* and *nat*. The first is responsible for deciding whether packets shall be bridged or

routed. The second offers MAC filter operations. The last table is used for Network Adress Translation (NAT).
Each table consists of several chains. These are illustrated in the right part of the upper level in Figure 2.7. The table association is shown by the colors around the boxes.
Packets traverse the *BROUTING* chain where *ebtables* decides whether a packet shall be routed or bridged. The default setting is bridging.
Afterwards NAT operations can be performed in the *PREROUTING* chain before the bridging decision is taken.
If the destination MAC of the current packet is used by a local NIC, the packet is passed to the upper layers of the network stack after it traversed the *INPUT* chain. The Forwarding Database (FDB) is the mapping between MAC addresses and ports. Each entry is associated with a timeout so that old entries are deleted after some time if they do not get updated. If no entry in the FDB exists, the bridge floods a frame to all its active ports. Through the flooding mechanism, the bridge learns the MAC addresses of NICs and to which port they are plugged in.
Otherwise the packets are flooded over all bridge ports. Therefore, packets go through the *FORWARD* chain of the *filter* table. This chain enables the creation of a firewall by using the MAC addresses or ports. Finally the packets go through the *POSTROUTING* chain and leave the NIC after they went through the *egress* traffic control system [12].

2.4.2 Traffic Control

Traffic shaping on Linux basically works by gaining control over a queue in which packets are enqueued. Each NIC has an incoming and an outgoing queue for packets. They are called *ingress* and *egress* respectively. An algorithm that manages a queue is called QDisc. The QDisc which is attached directly to the NIC is called *Root QDisc*. Such a QDisc can have configurable classes to handle provide different link impairments. It is called a *classful QDisc*. The inner elements are called *classes*. A *Handle* is used to identify QDiscs and *classes* [23]. It is a 32 bit field split into a major and a minor number. The major number of classes have to match the parent's QDisc minor number [38]. Filters can be used to redirect certain traffic to a class.
A *classless QDisc* provides no further inner division. The default QDisc is called *pfifo_fast* and applies only some QoS to the traffic by classifying traffic according to the Type of Service (ToS) flag of IP packets [23].

2.4.3 Linux Wireless Drivers Overview

Figure 2.8 shows the integration of a Linux wireless driver in the Linux system and the interaction of user-space programs with the kernel-space. First, a driver is loaded via the *insmod* system call (top left of the Figure) which may happen automatically during boot or manually. Moreover, the system call initializes the driver, afterwards the hardware (NIC) and register functions in the kernel-space used e.g. by the TCP/IP stack for the transmission, reception and management (*Transmit functions* and *Receive functions* in the Figure).
Packets are received and sent from user-space by the *send* and *receive* socket system calls. A user can now send packets by leveraging the send system call which passes the data to the TCP/IP stack. The stack adds the IP and TCP layers to the payload and uses the *Transmit path*

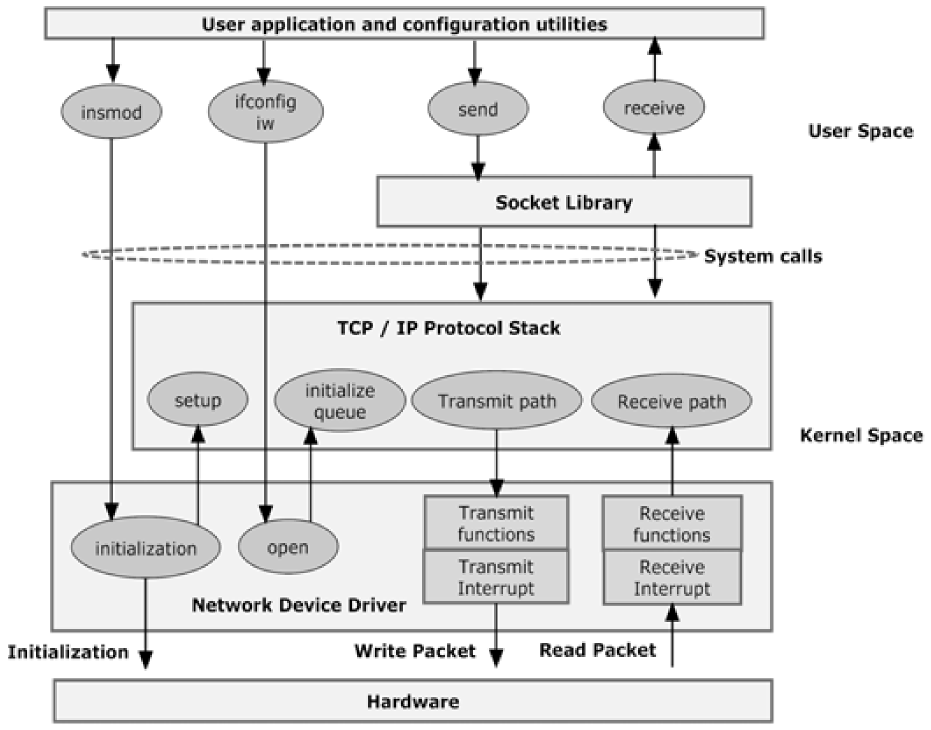


Figure 2.8: Integration Of A Driver In The Linux Kernel Network Stack [44]

to send the packet to the driver. The driver triggers a *Transmit Interrupt* and signals the NIC that a frame needs to be delivered.
The same proceeding applies then a frame is received: First an interrupt is triggered so that the driver can receive and process the frame from the hardware. In each step to the user-space, the corresponding layers are discarded from the PDU depending on the type of socket [44].
Finally the state of a NIC is configured by tools such as ifconfig and iw.
Figure 2.9 shows that these user-space tools interact either with the *nl80211* interface or the deprecated wireless-extension (wext).
Nl80211[5] is the new 802.11 interface based on netlink[6] which is used for communication between kernel and user-space via the socket facility and a new domain AF_NETLINK[7] instead of e.g. SOCK_STREAM.

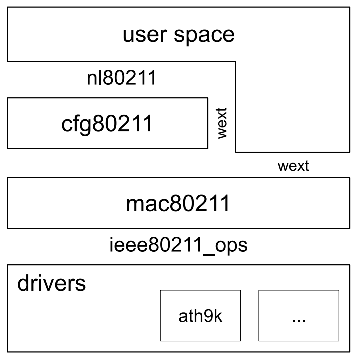


Figure 2.9: Linux Wireless Design [26]

nl80211 uses *cfg80211* which handles the configuration options for 802.11. This includes station management such as adding, removing, modifying and dumping stations. Scanning of APs, setting the wifi channel, bitrate and interface modes like the mesh mode are handled by *cfg80211* [44, 26].
Most new wireless cards do not implement the MAC layers inside the device itself. Instead,

[5]https://wireless.wiki.kernel.org/en/developers/documentation/nl80211
[6]http://www.ietf.org/rfc/rfc3549.txt
[7]http://man7.org/linux/man-pages/man7/netlink.7.html

they rely on *mac80211* to handle the MAC layer. This approach is called *softmac* and lowers hardware costs. For example 802.11 frame creation and rate control algorithms are implemented inside *mac80211*[26]. Wireless drivers interact with *mac80211* via the *ieee80211_ops* callback interface.

2.4.4 Relation To MiniWorld

MiniWorld uses *ebtables* in the **Bridged LAN** network backend to create a firewall based on the bridge ports. This allows MiniWorld to control with whom a NIC is allowed to communicate. Linux bridges are used for both **Bridged** network backends. Furthermore, TC on *egress* is used to provide basic link effects such as bandwidth and delay. The **WiFi** network backend leverages a Linux driver which simulates wireless NICs inside the Virtual Machines (VMs).

2.5 Virtualization

Virtualization is an essential technology nowadays. Computer science is all about abstraction and virtualization is just another kind of abstraction. It enables multiple OSs to run on the same hardware by providing the illusion to each system that it runs alone and has full control over the underlying hardware. Actually, each virtualized OS, also called VM, is run by a *hypervisor*.
The abstraction from the hardware to the hypervisor allows better resource distribution and usage. Multiple VMs can run on the same server without affecting each other, limited only by the availability of resources. Using one server e.g. to host an in-house chat software system on a server with much resources is not very clever and not cost-efficient.
Moreover, VMs provide security to a software which is security-critical. Even if an attacker can break into the Virtual Machine, he/she still cannot break out of the *hypervisor*, also called Virtual Machine Monitor (VMM).
In recent years it enabled Amazon WebServices etc. to provide Infrastructure as a Service (IaaS) clouds where infrastructure is provided dynamically in terms of VMs. Moreover, it enables the usage of elastic services and due to its abstraction it is much easier to migrate a VM than a process. This offers great opportunities for load balancing when resources are fully exploited on a server.
The aforementioned hypervisor-based virtualization approach is not the only one. Instead of virtualizing a full system and providing the illusion of a specific or the actual hardware to an OS, so called *Lightweight Virtualization* enables the separation of processes and is much more efficient but has the limitations of being only able to run software that is supported by the OS Application Binary Interface (ABI) and kernel.
Both approaches have their advantages and disadvantages.
MiniWorld uses full system virtualization whereas most related work makes use of process isolation. Both virtualization technologies are explained in the following.
Note that the following is basically based on Tanenbaum et al. [40].

2.5.1 Hypervisor Virtualization

Virtual Machine Monitors can be either integrated into an existing OS or run directly (bare metal) on the hardware. Figure 2.10 visualizes both approaches. The left side of the Figure

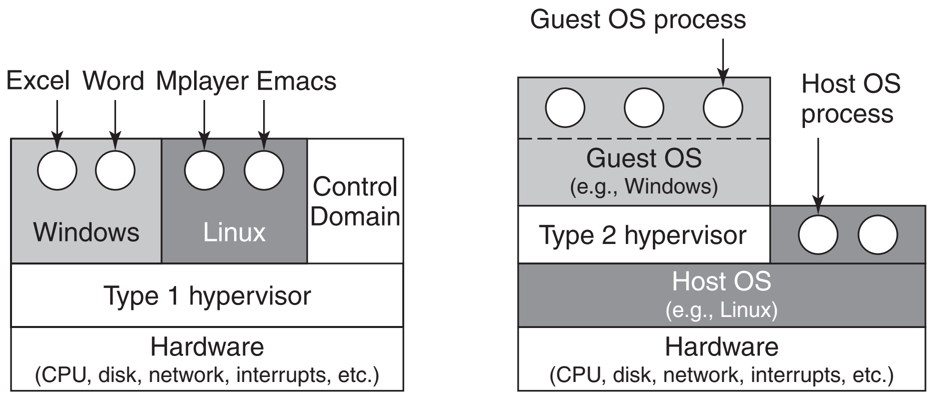


Figure 2.10: Type 1 Hypervisor (left side) vs. Type 2 Hypervisor (right side) [40]

shows a type 1 hypervisor. Xen, VMware ESXi, and Microsoft Hyper-V are examples of such a Virtual Machine Monitor. The Figure illustrates that the hypervisor is running directly on the hardware. This is in contrast to a type 2 hypervisor[8] shown on the right side of the Figure, there the VMM is running inside an Operating System, mostly as a user-space process combined with a kernel module. Therefore it is also called hosted supervisor. Both Figures show that full system virtualization enables to emulate arbitrary Operating Systems without the knowledge and modification of such a system.
The x86 platform is very hard to virtualize without support from the processor. This is due to the architecture where instructions have different semantics depending on the processors protection ring. These instructions are called *sensitive instructions* hereinafter. The x86 platform has four rings, where ring 0 is the most privileged and therefore the kernel operates inside it. The so called *kernel mode* allows any processor instruction to be executed. User processes are run in ring 3 called *user mode*. Hypervisors leverage the fact that the rings 1 and 2 are unused, therefore a guest operating system is running in ring 1. If the guest OS tries to execute a privileged instruction, a trap[9] is forwarded to the hypervisor which can perform some sanity checks and then decide whether it shall perform the instruction on behalf of the guest OS. The separation into different protection rings prevents other rings e.g. from accessing the kernel memory. Therefore the guest is not allowed to execute sensitive instructions. Note that this *trap and emulate* approach only works with recent processors which have a special virtualization extension[10], called VT in the following. With this extension, a set of operations which shall be trapped can be set with a hardware-bitmap by the Virtual Machine Monitor. E.g. I/O always traps because the guest has no direct access to the hard disk. A paper from Popek and Goldberg states that a platform is only virtualizable if the sensitive instructions are a subset of the privileged ones which is basically achieved with the virtualization extension [40].
Even without this special hardware support, hypervisors are able to virtualize the x86 platform. For that purpose, a technique called *binary translation* is used where the code of the guest kernel is rewritten so that sensitive instructions are replaced by a call to the hypervisor. The rewriting process makes use of a cache for already converted *basic blocks*[11]. Guest

[8]KVM, Virtual Box and Parallels are an example of type 1 Virtual Machine Monitors.
[9]A trap is a special kind of interrupt caused by an exception or fault.
[10]Secure Virtual Machine (SVM) for AMD, Virtualization Technology (VT) for Intel processors
[11]A basic block is a sequence of instructions ending with a flow control altering command such as jmp instruction

processes can be ignored at all because they are executed in an unprivileged protection ring. Additionally, there are many optimizations so that *binary translation* achieves a good performance.
Due to the protection rings and having a guest kernel unaware of virtualization which tries to execute privileged instructions, a trap handler has to be installed in the host kernel. Therefore, most type 2 hypervisors have a kernel module allowing it to operate in ring 0. The left part of Figure 2.11 shows the virtualization approach previously described (with

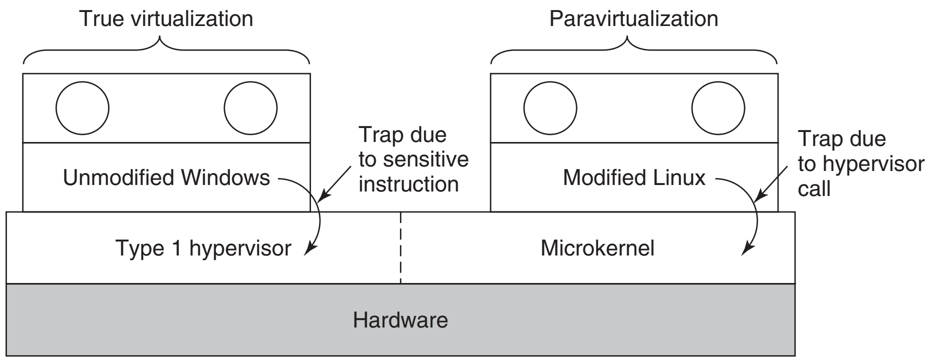


Figure 2.11: Full Virtualization vs. Paravirtualization [40]

or without VT). In the example there is a type 1 hypervisor, but of course a type 2 VMM can do true virtualization too. The right side of the Figure outlines another approach called *Paravirtualization*. Instead of providing the illusion that a guest OS is running on the real hardware, modifications to the kernel are done so that it cooperates with the Virtual Machine Monitor. *Hypercalls* are used between the paravirtualized guest and host kernel in the same manner a user-space application performs system calls to the kernel. The drawbacks of this approach are that the guest has to be modified but results in better performance on the other side [40].

KVM & Qemu

In the following the KVM and Quick EMUlator (Qemu) are depicted because MiniWorld uses KVM as a type 1 hypervisor.
KVM runs VMs as a normal user processes in Linux. A kernel module provides access to a character device at /dev/kvm. From a user-space perspective ioctl calls to the character device enable to start a VM, allocate memory for a VM, run a virtual CPU, read and write registers as well as to inject interrupts into the Virtual Machine. Note that KVM only works with VT enabled processors [25].
KVM works hand in hand with Qemu. In contrast to KVM, Qemu is able to emulate not just x86 CPUs but also PowerPC, ARM etc. There are 2 modes in which the emulator can run. The first is used for full system virtualization whereas the second is able to emulate a program compiled for a different processor architecture and is called *user mode emulation*. Moreover, it runs on all common platforms such as Linux, Windows and Mac. The android emulator is built on top of Qemu because it is capable of emulating the ARM CPUs [5, 25].
Qemu is a *dynamic translator*. This means that instructions for a different CPU are translated to the host CPU. The translation process uses a cache for performance reasons. Instructions are split into micro operations which are manually coded and compiled with GCC. The

translations of the micro operations are then cached. *Dyngen*, a part of Qemu can then create a dynamic code generator based on a sequence of micro operations as input.
Qemu is not only capable of emulating CPUs. It has many devices it can emulate such as NICs, mouse, keyboard, disks etc. [5].
KVM leverages Qemu to provide these device models to a guest VM.

2.5.2 Container Virtualization

In recent years *container virtualization* became more and more popular. Especially *Docker* came up in the cloud environment because it allows to ship an application in a container which requires no further setup or installation. Instead, developers can create containers for their software and upload them to the *Docker Hub*, a public repository for *Docker* images.
The ancestor of containers is the *chroot* command. It allows to restrict file system access to a specific directory which is then treated as the new root directory. The change root approach is not very secure but shows the first attempts to isolate processes in terms of file system access.
Container virtualization is all about isolating a process. In Linux, there are different *namespaces* which allow the isolation of a specific resources. E.g. *network namespaces* enables each container to have an own network stack where routing tables, ports and interfaces are not conflicting with the network stack of the OS outside the container [13]. The namespaces man page[12] lists additionally namespaces for Inter-Process Communication (IPC), mounts, PIDs, user, group IDs etc. They enable processes to be completely isolated from each other. Even different users can exist inside a container.
Containers additionally make use of *cgroups*[13] which allow processes to be organized into control groups so that resources can then be limited and monitored.
Despite *Docker* there is also Linux Containers (LXC) which enables the usage of containers on a lower level than *Docker* [13].
Most related work in the field of network emulators use *lightweight virtualization* whereas MiniWorld uses full system virtualization to provide more flexibility in what shall be tested. Moreover, *Docker* containers are used in an experiment in the evaluation Section.

[12]`http://man7.org/linux/man-pages/man7/namespaces.7.html`. Last viewed on 10.11.16
[13]`http://man7.org/linux/man-pages/man7/cgroups.7.html`. Last viewed on 10.11.16

3 Related Work

Related work of network simulators is discussed only briefly in Section 3.1. Network emulation is presented in Section 3.2. The Section starts by discussing a software switch with link impairment features which is used by a network backend in MiniWorld (**VDE**). Moreover, since most related work in the field of network and also WMN emulation, lack a real WiFi interface in their network emulation approaches, WiFi virtualization approaches are discussed in Section 3.3. MiniWorld's **WiFi** network backend is based on the WiFi virtualization solution depicted in Section 3.3.3.
Systems which incorporate node virtualization and network emulation are illustrated in Section 3.4.

3.1 Simulation

Simulation is a very old technique. Implementations make use of extra libraries, frameworks and even new simulation languages and Domain Specific Languages (DSLs) have been developed for discrete-event simulators. Examples of new simulation languages are Csim, Yaddes, Maisieand Parsec which are all derived from C and C++. Apostle and TeD are developments of Domain Specific Languages in this field [46].
New technologies often require new simulators for testing since research needs vary strongly. Therefore, simulators are not often reused and instead custom solutions have been developed. Moreover, very old simulators are often not available any more in terms of code or based on old technologies. In the following a short overview of simulators is provided.
Weingärtner et al. [46] provide a good overview of MANET simulators which are mostly from the last ten years. The most used simulator is *ns-2*. It is written in C++ and very precise physical layer models are available. Many extensions are provided by researchers and the community. It was primarily designed for wired networks, but extensions for well known MAC layers such as 802.11 and Bluetooth have been added. Mobility is provided by projects such as GEMM project, Graph Mobility project and the Obstacle mobility model. The simulator is written in the C++ language and requires scenarios to be created in the same language. The simulation can be controlled with the help of oTcl scripts which also define the network topology. The simulator lacks modularity and scalability is not that good. Therefore, a complete and independent new version called *ns-3* has been created.
Pdns can be used to boost ns-2 performance by distributing multiple ns-2 processes across the network. The network is divided into subnets which are then simulated on many cores.
OMNet++ is written in C++ and provides MAC layers for 802.11a/b/g and Bluetooth. According to Weingaertner et al. it is the most widely used commercial simulator [46].
DIANEmu aims at testing application protocols [22]. Other MANET simulators are *GloMoSIM*, *SimPy*, *JiST*, *QualNet*, *SWANS*, *GTNets*, *Jane*, *NAB*, *ANSim*, *Madhoc* and the *Jane simulator* [46, 22].

3.2 Network Emulation

In the following, network emulation approaches are presented. First *Netem*, a Linux QDisc is presented which offers certain link impairment functionality. Then, the *VDE* user-space switch is depicted in Section 3.2.2. It serves as the basis for MiniWorlds's **VDE** network backend. Afterwards, a SDN based switch called *Open vSwitch* is illustrated. Finally, WiFi virtualization approaches are discussed in Section 3.3.

3.2.1 Netem

Netem is a kernel module which is part of the Linux TC system. It offers link impairment functionalities via a QDisc. The link impairment includes packet delay, loss, duplication and reordering. Delays are defined by statistical distributions which are available via *iproute2*. Moreover, own distribution tables can be build from own test data. The loss of packets is controlled by percentage and a correlation value. Redundant routes can be simulated with duplicate packets. Bandwidth shaping is not included, but *netem* can be combined with the Tocken Bucket Filter (TBF) [18] or Hierarchical Token Bucket (HTB) QDisc.
According to the author of *Netem*, it has been built to validate the implementation of the *BIG TCP* and *TCP Vegas* algorithms inside the Linux kernel (2.6). The author explicitly claims that the internet is to complex to be simulated by *netem*, since it can not be depicted by a model [18].
Salsano et al. [39] provide a study of loss models and added more sophisticated loss models to *netem*. They integrated the *Gilbert-Elliot* and their proposed *4 State Markov-Model* to *iproute2* version 3.2.0.
Netem is used for link impairment by MiniWorld's **Bridged LAN** and **Bridged WiFi** network backends.

3.2.2 VDE

VDE is the acronym for Virtual Distributed Ethernet. It is meant to build a virtual network and supports hypervisors such as Qemu, Bochs and MPS. It is a distributed software ethernet switch running in user-space. Figure 3.1 depicts the architecture of VDE. There are 2 machines. The first is a UML instance, the second a Qemu node. Both are connected to a switch which is interconnected via a *VDE_PLUG* over the internet.
The software packages consists of the programs *vde_switch*, *vde_plug*, *wirefilter* etc. The first is the virtual switch, the second ties multiple switches together and the third enables network emulation.
The switch can operate as switch or hub. It has several ports, where so called cross cables can be plugged in with the help of *vde_plug*.

vde_plug uses a so called *dpipe* which redirects the stdout of one program to the stdin of another and vice versa. Therefore, it is a bidirectional pipe. Real computers can be integrated with the help of tap sockets, plugged into a switch port. Because *vde_plug* uses pipes, the switches can be connected e.g. via ssh or netcat. They only need to write the data to stdout. This design allows the program *wirefilter* to emulate packet loss, delay, duplicate packets, bandwidth, packet queue length, corrupted bits per packet and packet reordering by simply modifying the data read from stdin of another *vde_plug*.
Both *vde_switch* and *wirefilter* can be controlled via a Unix Domain Socket (UDS).

An article in the open-mesh wiki[1] proposes a patch to VDE called *color-patch*. Their intention is to use VDE together with Qemu to evaluate the B.A.T.M.A.N. routing protocol.
The patch build for VDE version 2.3.2[2] transforms the virtual network build by VDE, which ends in a single broadcast domain, into a one-hop-network. This means that packets can only travel one hop, if 2 switches are interconnected. Each port has a color. Packets are only allowed to be forwarded if the colors between the 2 ports differ. An example shows that the VM which is connected to the switch can have color 1 while other interconnecting ports have color WiFi. Therefore a packet from host A to B can only be forwarded on this path [11, 43].
Britos et al. [7] presented 2015 a network emulator called *BAMNE* which leverages the previously described color-patch and VDE together with VirtualBox. The emulator is tailored at performing tests with B.A.T.M.A.N.. The paper lacks experiments which show the scalability and performance of the solution.
The approach is not new. A paper from 2011 already showed first experiments using the aforementioned solution to evaluate the overhead of B.A.T.M.A.N., but used Qemu instead as the hypervisor [15].

3.2.3 Open vSwitch

In the Software-Defined Networks (SDN) approach, switches are controlled by a well defined interface called OpenFlow [33].
There are hard- and software based SDN switches that implement the OpenFlow protocol. The de facto standard software SDN switch is open vSwitch. Figure 3.2 shows the architecture of Open vSwitch. It is part of the recent Linux kernel. Below the dashed line of the Figure, there are 2 components of the switch: *ovs-vswitchd* is the user-space program responsible for handling packets based on rules (so called flow tables) which are inserted by a controller (upper part of the Figure).
The latter component is the *Kernel Datapath* which receives the packets from the OS NICs. The first time a packet is received it is redirected to *ovs-vswitchd* which has a set of actions and decides whether the packet shall be modified, dropped, routed etc. The *Kernel Datapath* serves as a cache for the decisions of the *ovs-vswitchd*. Since open vSwitch is a multi-layer switch, the packet classification allows a controller to match against any protocol field. Each rule has also a priority which is necessary if multiple rules match.
Durable settings of the switch such as adding or removing ports of the switch, configuring QoS queues or enabling the STP protocol are stored in the *ovsdb-server* (left part of the Figure) and can be configured by a OpenFlow controller via the *OVSDB* protocol.

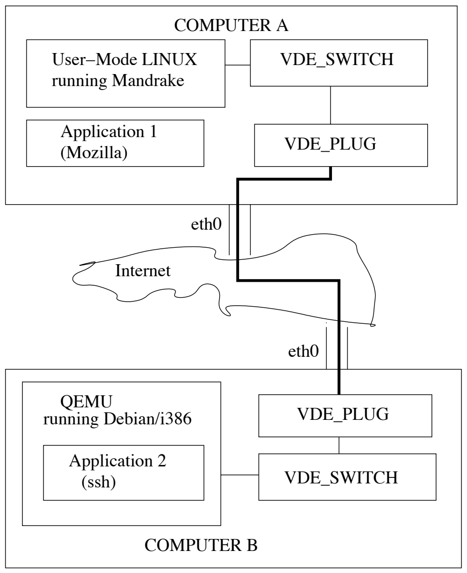


Figure 3.1: VDE Architecture [11]

[1]https://www.open-mesh.org/projects/open-mesh/wiki/Emulation
[2]http://www.open-mesh.org/attachments/download/152/vde2-2.3.2_colour.patch

Pfaff et al. demonstrate that open vSwitch is very fast. In a comparison against the Linux bridge, it achieves identical throughput in the simplest configuration of open vSwitch while having more CPU overhead (161% vs 48%) with a throughput of 18.8 Gbps. Adding a flow to open vSwitch and a corresponding iptables rule to the bridge shows that the CPU usage of the bridge increases to 1,279% while the CPU usage of open vSwitch remained constant [34].
Conclusion: The open vSwitch may be a good alternative to the Linux bridge, especially when much flexibility on the packet classification is needed. There needs to be done further research if link emulation can be achieved with the switch.
Moreover, a comparison between the wireless mode of MiniWorld and open vSwitch is needed in terms of packet classification. The wireless mode in MiniWorld uses ebtables to control which nodes can see each other. A controller in open vSwitch can probably replace ebtables.

3.3 WiFi Virtualization

All related work except *Mininet-WiFi* lack the simulation of a real WiFi card inside the virtualization layer. Xia et al. [48] point out that there are currently 2 approaches used for WiFi virtualization which are illustrated in Figure 3.3.

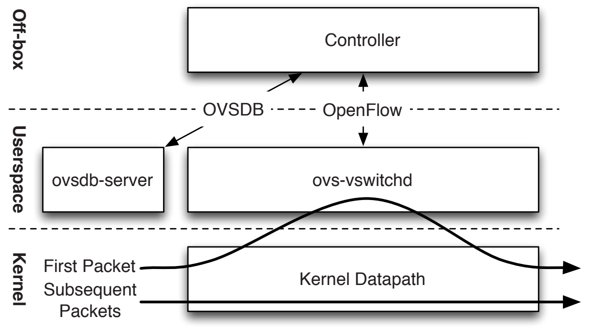
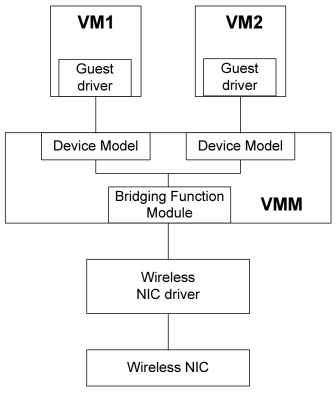
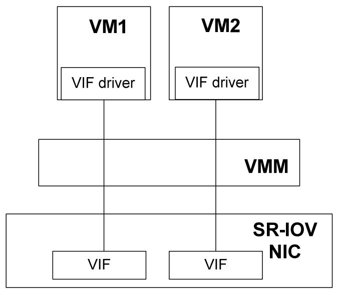


Figure 3.2: Open vSwitch Architecture [34]

The left side shows the software-based, the second the hardware-based approach. In the software-based approach there is only one wireless network card. The hypervisor takes care of providing each VM access to the WiFi card but emulates an 802.3 ethernet device inside the guest. Either the standard NIC drivers are used inside the guest or paravirtualized drivers are provided for performance reasons. Therefore, the VMs have no access to the actual wireless NIC.
In the second approach the virtualization is handled by the wireless NIC itself. Single Root I/O Virtualization (SR-IOV) is a specification allowing PCIe devices to expose multiple Virtual Interface (VIF) which can be directly assigned to a VM.
Xia et al. propose a combined approach called *virtual WiFi*. They argue that a para-virtualization approach where the hypervisor vendors provide the para-virtual driver is not feasible because the management interface between the driver and the wireless NICs is often proprietary. Therefore, hypervisor vendors could only provide the smallest common denominator.
The argumentation is true for a real wifi card whose functionality shall be exposed to multiple VMs, but does not hold for a virtual radio on the host which could be exposed via para-virtual drivers to the guests.
MiniWorld already has a 802.11 backend built-in, but this requires a user-space application to be run inside the guest. An approach which does not require VM image modification is to move the functionality to a Qemu device driver. Xia et al. also implement a device driver for Qemu, therefore the summary of their *virtual WiFi* is very interesting for the future



(a) Software-based approach



(b) Hardware-based approach

Figure 3.3: WiFi Virtualization Approaches [48]

development of MiniWorld.

3.3.1 Combined Software And Hardware Based Approach

Their prototype implementation is based on KVM and relies on an Intel's 5000 or 6000 WiFi card. Figure 3.4 illustrates the architecture of their approach which uses a combination of software- and hardware-based virtualization.
The bottom of the Figure shows the actual WiFi device whose microcode needs some modifications. In the host lives an augmented WiFi driver. On the top part of the Figure the VM to which the WiFi card shall be exposed is shown.
KVM can intercept requests from I/O ports. If the VM wants to access these ports, a VM exit instruction lets the processor switch control from the VM to the host. The I/O requests are then forwarded to Qemu used by KVM for the provisioning of virtual devices.
They developed a *Virtual Wifi Device Model* for Qemu which establishes an ioctl interface to the *Augmented Wifi Driver* during initialization to get a VM-ID. This is used to tag packets as well as commands from the VM so that the *Augmented Wifi Driver* can demultiplex commands/packets from the VMs. The *Augmented Wifi Driver* is a modification of the Intel driver to support the ioctl interface. The *WiFi Device*s microcode has to be aware of the VMs. Therefore a mapping between MAC addresses and the VM ids are allocated. The *Augmented Wifi Driver* thus knows about the VM id both for packet transmission as well as for packet reception [48].

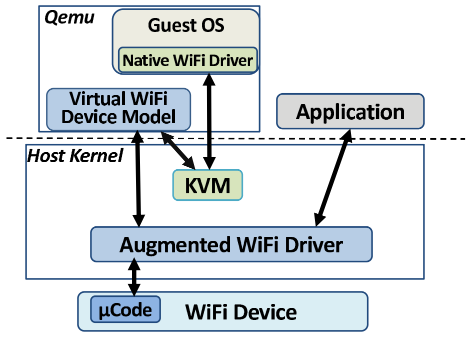


Figure 3.4: Virtual WiFi Architecture [48]

3.3.2 Qemu Atheros NIC Model

Another paper from Keil et al. [24] shows the creation of a virtual 802.11 device for Qemu. Unfortunately the code is only available for Qemu version 0.9.1 and has never been merged into the Qemu code base. Keil et al. devloped a wireless fuzz-test suite. They argue that device drivers are a security critical component in OSs.
Their virtual network device is based on the rtl8139 and ne2000 NICs which simulates an Atheros AR5212 802.11 a/b/g chipset. The driver they used is MadWiFi together with Ubuntu 6.06.
Keil et al. point out that they reverse-engineered the AR5212 chipset to create the virtual device model for Qemu. The device model interacts with the WiFi NIC through shared memory [24].

3.3.3 Mac80211_Hwsim & Wmediumd

There is a very interesting module called *mac80211_hwsim* which is shipped with a standard Linux kernel. It is able of simulating arbitrary numbers of 802.11 radios.
MiniWorld leverages this kernel module to implement a 802.11 network backend. Therefore, the functionality of a device driver as well as the integration into the Linux wireless stack is shown in the following. Afterwards the 802.11 simulator is explained in more depth as well as *wdmediumd* which simulates the wireless medium for *mac80211_hwsim*.

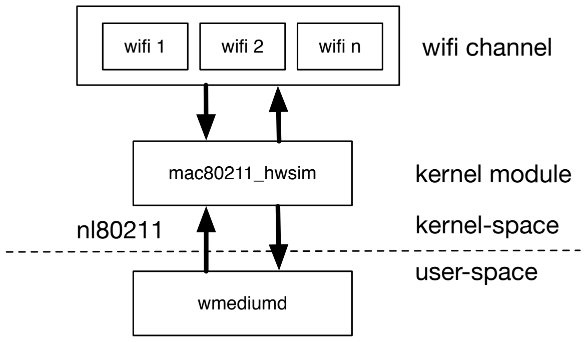


Figure 3.5: Wmediumd architecture

Further research showed that MiniWorld is not the only full system based virtualization approach which makes use of *wmediumd*. A custom build solution has been used to propose a Consensus Transmit Power Control (CTPC) algorithm for Wireless Mesh Networks [49].

The kernel module called *mac80211_hwsim* has been created 2008 and is intended to ease the development of Linux network tools such as hostapd and wpa_supplicant[3].
It works by distributing all frames from WiFi devices which are on the same channel among them.
mac80211 views *mac80211_hwsim* just as another hardware driver like *ath9k* is in Figure 2.9. The kernel module has a parameter which defines the number of radios which shall be emulated. Moreover, a device in monitor mode is created [29] which enables to view the unaltered 802.11 frames plus information about the wireless system such as signal to noise ratio, and the modulation scheme. These extra information are encapsulated in the *radiotap* header [44, 26].
Therefore the wireless simulator uses the Linux 802.11 wireless stack as well as the remaining network stack. This enables experiments with the simulator to use a real 802.11 implementation. Note that the WiFi simulator does not simulate the wireless channel.

[3]The first is a daemon running in user-space which enables the creation of 802.11 access points. The latter is an authenticator for WPA/WPA2/802.1X etc.

Wmediumd[4] adds these capabilities to the simulator. Figure 3.5 shows the architecture of the C software which runs in user-space (bottom of the Figure). There is a netlink API inside the simulator which allows to capture the 802.11 frames. *wmediumd* uses this API and adds packet loss and delays with random backoff based on a configuration which defines statically the signal strength between peers. The frames are sent back to the kernel via the same interface.

3.4 Emulation Systems

The first network emulation systems which is discussed verbosely is the Common Open Research Emulator (CORE) since the **Bridged LAN** and **Bridged WiFi** network backend rely on approaches which CORE introduced (Section 3.4.1).
A paper from To et al. [42] discusses the integration of the network simulator *ns-3* into a network emulation system with Linux bridges (Section 3.4.3). The approach could be implemented by MiniWorld too, to provide a network backend with high fidelity simulation.
To the best knowledge of the author, *Mininet-WiFi* is the only network emulation system which provides a real WiFi NIC to its virtual nodes. The system is presented in Section 3.4.5 to the reader.

3.4.1 CORE

The Common Open Research Emulator (CORE) is a fork of the Integrated Multiprotocol Network Emulator/Simulator (IMUNES) developed by the Boeing research and technology devision [8] and further supported by the Naval Research Laboratory.
It is open source and written in python. MiniWorld is heavily inspired by CORE. One very important point in which MiniWorld differs is that it uses full system emulation whereas CORE uses container virtualization technology.
Note that CORE comes with a graphical User Interface (UI) which allows to define network topologies. They can be exported to XML files and used to switch between topologies in the **CORE Mobility Pattern** of MiniWorld. In the following, the architecture of the emulator is illustrated.

Architecture

CORE uses containers to isolate nodes from each other. It supports FreeBSD as well as Linux.
All nodes share the same kernel since container virtualization is used. Therefore protocol stacks other than the ones which are present in the kernel of the host machine, can not be emulated. Further limitations are that router images, different OSs and different kernels are not usable with CORE as well. The decision to support only ABI matching applications is a tradeoff between performance and flexibility. MiniWorld in contrast, supports every OS and application which runs under Qemu.
CORE uses OpenVZ and Linux namespaces to achieve container-based process isolation under Linux.
For FreeBSD it uses jails. Since the network emulation and distributed emulation differs for each supported OS, they are outlined separately from each other in the following.

[4] `https://github.com/bcopeland/wmediumd`

FreeBSD

The first version of CORE supported only FreeBSD. CORE leverages the BSD Netgraph system to create a certain network topology. Ahrenholz et al. [1] showed that the Netgraph system uses the available processor cores much better and is able to achieve a much better throughput in terms of packets per second compared to the Linux network backend. Netgraph was able to deliver more than 2 times more packets per second due to zero-copy semantics achieved by passing packets by reference only. Therefore, time-consuming memory copies are prevented [1].
The network topology creation of wired nodes can be easily achieved with the Netgraph system. To mimic the wireless channel, they utilize the Netgraph hub node, which simply forwards all packets to any connected node except the originator. Ahrenholz et al. added a hash table to the hub node which controls, based on source and destination node id, the link quality between peers.
Link effects are applied by letting the nodes connect via a pipe to the Netgraph system. The effects can then be applied to the pipe on a per packet basis depending on the source and destination.
For more complex link effects, CORE uses a modular C daemon which receives node information from the GUI and calculates link effects such as bandwidth, delay, loss, duplicates and jitter. Afterwards the wlan kernel module is configured via the libnetgraph C API.
Ahrenholz et al. mention a simple wireless model where the delay and loss increases when the distance between nodes increases as well.
Hence, the link effects are controlled by the distance between nodes like in MiniWorld.
A simulation can also be distributed among several emulation servers. A daemon written in C (*Span*), creates TCP/IP tunnels for the nodes. The sockets are Netgraph nodes too, therefore they can be connected to a CORE node.
For wireless scenarios, Netgraph kernel sockets (ksockets) enable connecting to another machines kernel and appear also as a Netgraph node.
For packets sent to a ksocket, the hash table lookup is only done at the receiving side. To perform the demultiplexing, packets are prepended with the source id [2].

Linux

There are three different ways to setup the virtual networks for the Linux platform. The first uses Linux bridges and netem. The second leverages Extendable Mobile Ad-hoc Network Emulator (EMANE) and the last integrates with the real-time scheduler of ns-3 [1]. Each possibility of network emulation is depicted in the following:

- **Linux Bridges + netem**: Linux bridges mimic a hub/switch in the form of a NIC in the OS. Netem is a kernel module which is able to apply basic link effects (bandwidth, delay, loss, etc.) based on filters.
  Both OpenVZ and Linux namespace containers use the *veth* driver which provides a pair of NICs. One resides in the container, the other lives on the host and is bridged together with nodes which share connectivity. Note that this needs one bridge per connection.
  For wireless connections, CORE uses *ebtables* to configure which node can reach any other node. Ebtables is the layer 2 equivalent of iptables. Rules are based on the MAC layer [1].

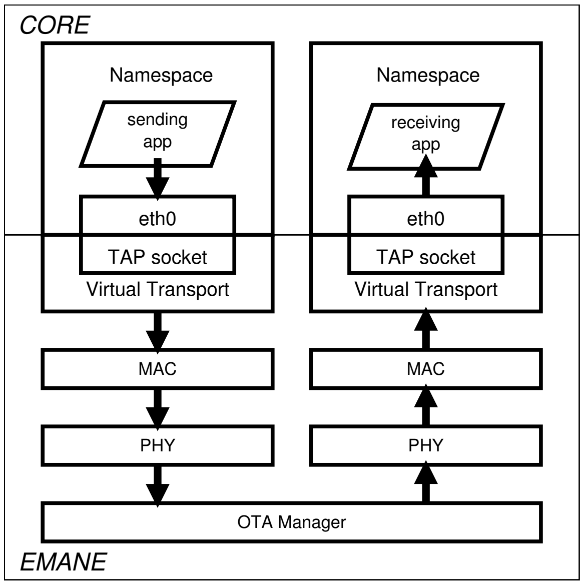


Figure 3.6: CORE EMANE Integration [3]

- **EMANE**: The network emulator provides high fidelity link emulation with pluggable PHY and MAC layers. EMANE could also be integrated into MiniWorld, therefore only its integration is illustrated in Figure 3.6. The top part of the Figure shows 2 CORE nodes living in a network namespace. The bottom part shows that EMANE can emulate the PHY and MAC layers. Packets are delivered by the over-the-air (OTA) Manager with the help of multicast. In the middle of the Figure one can see *eth0* and *TAP socket* which tie CORE and EMANE together. A tap device has 2 parts, one in user-space, the other in kernel-space. The user-space part is the *TAP socket* and controlled by EMANE. Packets sent by the CORE node are read from the tap device socket and delivered by the OTA manager. Moving nodes in the UI generates EMANE location events [3].
- **Ns-3**: The discrete-event network simulator provides high fidelity network models such as 802.11, WiMAX, LTE etc. Furthermore, it features several mobility models. Because ns-3 runs in simulation-time and not in wall clock time, the real-time scheduler of ns-3 has to be used.
  The integration also uses a tap device. There is not yet support for ns-3 in the GUI. Hence, ns-3 has to be scripted with the help of the ns-3 python API [10].

Distributed Mode The distributed mode in Linux uses Generic Routing Encapsulation (GRE) tunnels[5] to connect nodes living on different emulation servers. Although there is no auto-distribution. The consequence is that nodes have to be manually assigned to a specific emulation server. This is different in MiniWorld where a scheduler carries out the node assignment task according to resources of the emulation servers.

[5]Gretap to be precise

Summary & Comparison

CORE has been illustrated very detailed because MiniWorld is very familiar with CORE. MiniWorld includes the bridge + netem backend and supports both the wired as well as the wireless connection mode (MiniWorld's **Bridged WiFi** and **Bridged LAN** network backend). Moreover, MiniWorld also uses GRE to enable node communication among emulation servers. The EMANE and ns-3 integration is very interesting and could be ported to MiniWorld. The benefits for MiniWorld are more precise PHY and MAC layer as well as more mobility patterns. Furthermore, MiniWorld utilizes the scenario editor of CORE and provides the possibility to change scenarios in predefined time steps (MiniWorld's **CORE Mobility Pattern**).
To sum up, CORE supports FreeBSD (jails) and Linux (OpenVZ, Linux namespaces). Either static topologies can be used or nodes can be moved dynamically by a ns-2 mobility script. Both wired and wireless connection modes are supported. Wireless communication is emulated with the help of netem, ns-3 or EMANE. The fidelity of models range from a basic on-off model to very precise emulation of PHY and MAC layers.
The papers from Ahrenholz et al. ([2, 1, 3]) lack a study of scalability, especially for the distributed mode. They claim that CORE supports a few hundred nodes. The lightweight virtual machines scale very well, but if CORE also does, is not proven.

3.4.2 IMUNES

CORE started as a fork of IMUNES in detail. The authors of IMUNES added network stacks virtualization to BSD. They created a new kernel structure called *vnet* and placed static and global variables there so that a network stack could be created multiple times without affecting each other.
Moreover, they enabled virtual nodes to communicate with each other without inducing additional performance overheads by leveraging the unique BSD Netgraph system which passes packets by reference.
The paper [50] mentations the BSD link emulators ng_pipe and *dummynet* for network emulation. In contrast to CORE, IMUNES only supports the BSD Operating System.

3.4.3 Dockemu

The authors of [42] argue that existing emulation solutions do not take the time-consuming step of installation and configuration into account when it comes to the software a researcher wants to evaluate. Therefore, they propose Docker containers as the virtualization layer. Docker is a very new technology which allows shipping applications in a box. So called Docker Images are self-contained and very popular in the DevOps field, especially in cloud computing.
Dockemu utilizes Linux bridges together

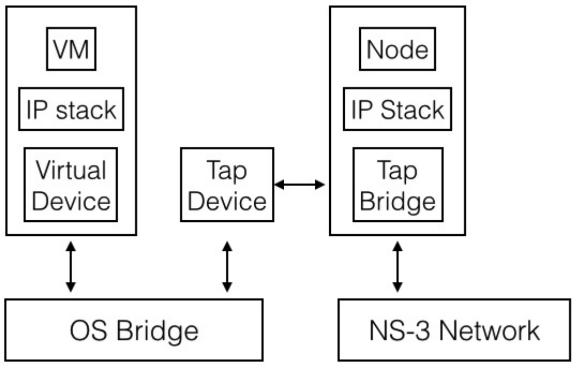


Figure 3.7: Ns-3 Tap Bridge Model [42]

with ns-3 to emulate PHY and MAC layers. Figure 3.7 shows how ns-3 integrates into Dockemu. The top left part of the Figure shows the Docker container with a veth device added to a Linux bridge. The right part of the Figure depicts the ns-3 network. A tap device is added to the bridge on the OS endpoint. Inside ns-3 the tap socket is managed by the *Tap Bridge*.
The architecture outlined in the Figure enables the integration of a network emulator within ns-3. In place of the veth device, one can also use a second tap device. For MiniWorld e.g., the tap device created by Qemu could be added to the *OS Bridge*.
The tool is written in python and bash. It is CLI based and controlled by a config file. It comes with OSLR and BMX6 as routing algorithms if one is needed. Other parameters such as the emulated layer and a mobility pattern can be defined there [42].
Comparison: MiniWorld uses KVM instead of *Docker* containers as the virtualization layer, but adding *Docker* to MiniWorld would increase performance for scenarios where full system virtualization is not required. Moreover, routing is not built into MiniWorld. Instead connectivity is provided on the link layer. Moreover, no simulation is used by MiniWorld's network backends.

3.4.4 Mininet

Mininet is an emulator for SDN and comes with *open vSwitch*. It[6] uses Linux network namespaces and process-isolation for each emulated node. There is one veth device pair for each connection. One part of the veth pair is put into the namespace, the other is connected to the *openflow switch* [45]. Controllers can run on the real network or inside the network namespaces as long as they are reachable by means of TCP. There are a couple of network topologies which can be created via the Command Line Interface (CLI). *Mininet* ships with a VM which has useful tools, e.g. Wireshark and dpctl which can control and view the flow tables of a OpenFlow switch. The CLI enables an easy control over the network namespaces. Commands prefixed with a specific node id are executed in the appropriate namespace [33].

3.4.5 Mininet-WiFi

There is a fork of *Mininet* called *Mininet-WiFi*[7]. Fontes et al. use the *mac80211_hwsim* Linux kernel module to simulate arbitrary numbers of 802.11 devices. Each virtual node still uses a veth device pair for the communication between the host and the virtual nodes. On the host a number of 802.11 radios is simulated with the help of mac80211_hwsim. Each wifi device is then bridged to an OpenFlow capable AP switch. Their tool leverages hostapd and wpa_supplicant to create Stations (STAs) and APs. Furthermore, many mobility models such as GaussMarkov, RandomDirection, RandomWalk, RandomWaypoint and TruncatedLevyWalk are supported [14].
The tool can be used via CLI, API or the Visual Network Description GUI which is capable of generating *Mininet-WiFi* code.

[6]`http://mininet.org`. Last viewed on 02.11.2016
[7]`https://github.com/intrig-unicamp/mininet-wifi`

3.4.6 Netkit

Netkit uses a different virtualization approach than the related work seen before. It is a mix between full-virtualization and container-based isolation. User-Mode Linux (UML) is a port of the Linux kernel designed to run as a user-space application. A UML VM is comparable to existing full-virtualization software in the fact that it supports the emulation of arbitrary devices.

Figure 3.8 depicts the architecture of Netkit. The upper part of the Figure outlines the UML nodes. They are interconnected via the tap module with a virtual hub (middle of the Figure). The virtual hub can also be connected to the Host kernel, allowing to integrate remote applications and providing internet to the VMs.

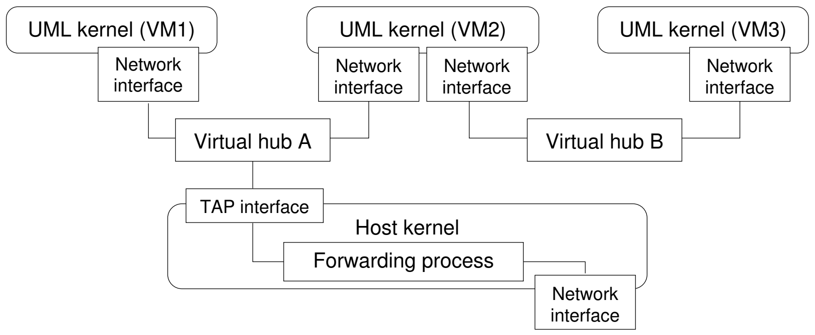


Figure 3.8: Netkit Architecture [35]

Netkit comes with a default file system image for the VMs which is a Debian Linux. All nodes share the same file system for reading, while they use a Copy On Write (COW) mechanism to create a write-layer on a per-node basis. This reduces the amount of disk space needed for the image [35].
To the best knowledge of the author, there is no scenario file format. Instead, topologies are created via the Command Line Interface.
Comparison: Netkit is built for educational purposes and seems to lack many features such as link emulation or a wireless mode at all.
MiniWorld also has a network backend that makes use of a software switch called VDE. The switch uses also tap devices to interconnect the nodes but offers link emulation properties such as bandwidth, delay etc. Moreover, MiniWorld utilizes the COW approach in the same manner like Netkit does.

3.4.7 Cloonix-net

Cloonix-net is a modification of the Cloonix emulator[8] whose changes have been integrated into the Cloonix distribution. Cloonix-net is very similiar to Netkit, because it uses UML as the virtualization layer. It adds the possibility to run multiple UML kernels and file systems to Cloonix. Moreover, it leverages netem to provide basic link emulation features. Rehunathan et al. present a detailed study of the Mobile IPv6 as well as the NEMOv6 protocol [37].

[8]`http://www.clownix.net/`

3.4.8 GNS3

Li et al. summarize the open source GNS3[9] emulator capable of emulating Cisco routers. The emulator was built to help people prepare for Cisco exams.
Cisco routers have a special OS called *IOS*. Graphical Network Simulator 3 (GNS3) includes an own hypervisor called *Dynamics* which is also the main simulation engine. The virtual machine monitor allows the simulation of multiple nodes on multiple servers by interconnecting the hypervisors via TCP/IP. GNS3 can only emulate Cisco routers, not switches [27].
Besides CORE, it is the only simulator with a distributed mode.

3.4.9 NEMAN

NEMAN is a network emulator which is built for the special goal of testing middleware and application layer protocols. Figure 3.9 outlines the architecture of the emulator. The most important component in Network Emulator for Mobile Ad-Hoc Networks (NEMAN) is the *Topology Manager* which is responsible for network switching and the creation of the network topology. The *Processes* (lower part of the Figure) are the actual applications a researcher wants to evaluate. They are connected with the *Topology Manager* by means of tap devices. Therefore each application needs to be modified to bind to the tap device[10].
Frames sent via the tap device are demultiplexed by the *Topology Manager* and switched according to the topology information. There is a *monitoring channel* called tap_0 in the Figure, which is interconnected bidirectionally to all tap devices. This enables the use of standard tools such as tcpdump for monitoring and analysis purposes. Moreover, due to its bidirectionality, the monitoring channel allows to induce packets into the network.

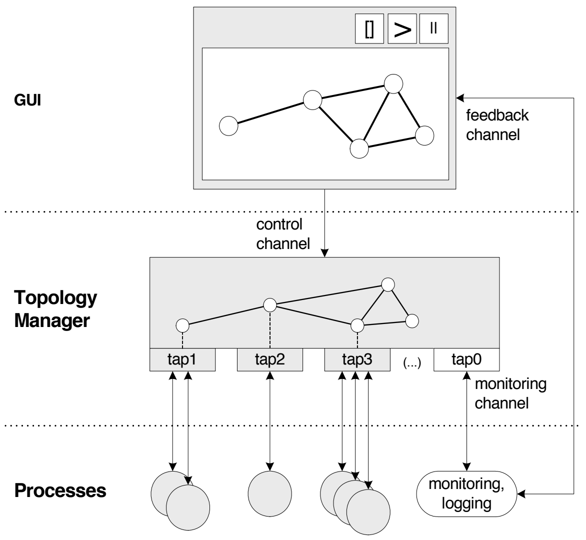


Figure 3.9: NEMAN Architecture [36]

Topology information are sent from the GUI (upper part of the Figure)[11] via a UDS. It is called the *control channel*. The GUI illustrates the network topology and visualizes feedback from the processes (lower part of the Figure). For this purpose it parses the output of the OSLR daemon used for routing on an IP basis.
The authors claim that NEMAN is able to handle a few hundred nodes on a single machine, but show only an experiment with 100 nodes.
The tool relies on ns-2 scenario files for the description of mobility and topology. Note that NEMAN does not apply link effects. Ns-2 is not used for networking, only for the scenario format.
An interesting feature is that the emulator allows to induce events at certain times. The

[9]`https://www.gns3.com`
[10]With the SO_BINDTODEVICE socket option
[11]The GUI is derived from the MobiNet GUI and used Tcl/Tk

experimental evaluation points out some problems whose solution requires a kernel patch [36], but may be included in a recent Linux kernel as the paper dates back to 2005.
Comparison: The approach of NEMAN requires binaries to be modified, in contrast to MiniWorld. There is no virtualization layer at all since processes simply bind to a tap device. Moreover, there is no link impairment.

3.5 Summary

Related work in the field of network emulation has been presented. *Mac80211_Hwsim* together with a modification of *Wmediumd* builds the basis for MiniWorld's **WiFi** network backend. *Open vSwitch* may be a good alternative to Linux bridges, but further research is needed if the SDN switch provides link impairment itself or if the Linux TC system has to be used.
MiniWorld's **Bridged LAN** and **Bridged WiFi** network backends are based on the network emulation approaches of CORE. Moreover, CORE scenario files can be built with the CORE UI and used by MiniWorld's **CORE Mobility Pattern** to switch between toplogies.
Container virtualization such as Docker could be used as additional virtualization layer for scenarios where full system virtualization is not required since it does not introduce additional emulation overhead.
There is no high fidelity link emulation, hence, MiniWorld could be combined with *ns-3* as described by CORE or *Dockemu*.

4 Design

This chapter introduces MiniWorld to the reader. Initially, the abstract design is filled with implementation details (Section **??**). a short overview and the design goals of MiniWorld are presented in Section 4.1. The overall architecture is depicted in Section 4.2. Afterwards, the typical workflow with MiniWorld is described in Section 4.3. Following is the introduction of the *Scenario Config* in Section 4.4. The different interfaces a node can have, are discussed in Section 4.5. The process of starting nodes and the integration of a network backend within MiniWorld is illustrated in Section 4.6. The *REPLable* mechanism is presented in 4.7. Link quality models are discussed in Section 4.8. Movement pattern are depicted in 4.9. Finally, the architecture of all 4 network backends is presented (4.10), followed by the distributed mode (Section 4.11).

4.1 Short Overview & Design Goals

MiniWorld is a **network emulator framework**. It provides great **flexibility** because it allows to emulate arbitrary programs and systems. This flexibility is traded by performance due to the usage of full system virtualization. Therefore, MiniWorld is neither limited to emulating Linux binaries nor to emulating Linux VM images even though MiniWorld itself only runs on Linux.
Full system virtualization poses high demands on resources, but MiniWorld is nevertheless build with **performance** in mind.
A *Snapshot Boot Mode* enables VMs to boot from snapshots and reduces the time needed to start the Virtual Machines drastically. Especially the switching of network topologies has been studied extensively, allowing to switch huge topologies in a few seconds. To remove the bottleneck introduced by *full system virtualization*, MiniWorld allows every Linux OS to participate in the emulation. In particular, network communication in the distributed mode is very fast to reduce possible bottlenecks which may arrive from the coordination of clients.
MiniWorld's goal is to be a network emulation framework. Therefore, **modularity** and **exchangeability** are very important design goals. Network backends, link quality models and movement patterns can be exchanged easily. MiniWorld comes with 4 network backends, 4 mobility patterns and 3 link quality models. Despite MiniWorld's goal to be a framework, it is usable without any further modifications and/or extensions.
Finally, **transparency** is one of MiniWorld's design goals. Network emulators are mostly complex systems, making debugging and traceability very hard. Therefore, the important steps in the emulation process such as booting the VMs and especially switching to another network topology are logged to stdout as well as to log files. The commands needed to create a particular network topology are written transparently to a log file. This enables users of MiniWorld to extract the topology and get a better idea of how network switching is done in the particular network backend. Summing up, the design goals of MiniWorld are as follows:

- Performance (Concurrency, Distribution)
- Flexibility (Virtualization)
- Modularity & Exchangeability (Framework)
- Transparency (Network Switching and Shell Command Logs)

In the following the typical workflow of MiniWorld is presented. This gives the reader a better understanding of what MiniWorld is and how it works.

4.2 Architecture

The architecture of MiniWorld is depicted in Figure 4.1. An **Analyst** requires 2 things to

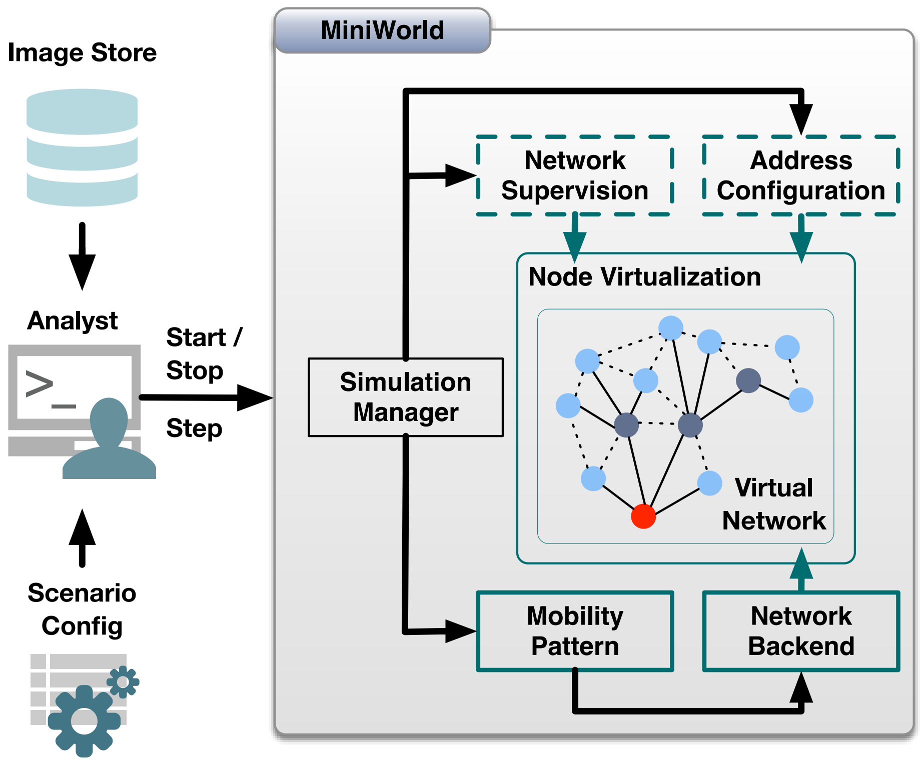


Figure 4.1: MiniWorld Architecture

start a simulation: A **Scenario Config** and an image from the **Image Store**. The config summarizes all information that are required to start the simulation and to setup the **Virtual Network**. An *image* bundles the software an *Analyst* wants to evaluate.
The simulation lifecycle is handled by the **SimulationManager**. This includes starting and stopping of a scenario. A **step** changes the topology of the *Virtual Network* and requires a distance matrix from the **Mobility Pattern**. The matrix contains the distances between any nodes. *Stepping* can be performed by the user or automatically by the *SimulationManager*. The network topology is changed by the *Network Backend*. The **Virtual Network** consists of nodes which are provided by means of virtualization. The *Network Backend* (impairment model) can operate either in user-space or kernel-space and is responsible for the link setup and impairment. The impairment scenario may be static, trace-based or event-driven. **Address Configuration** and **Network Supervision** are optional elements which are incorporated

into the design of MiniWorld. The first enables nodes to communicate with each other by means of addresses, the latter supervises the network setup of the *Network Backend*. The *Network Supervision* feature proves that the network topology has been setup correctly so that experiments which are conducted within MiniWorld are reliable. Moreover, it eases the integration of new *Network Backends*.
All turquoise depicted rectangles in Figure 4.1 are per design exchangeable. The abstraction which is provided by the *SimulationManager* enables the *MobilityPattern* and *Network Backend* to operate independently of each other.
The 2 design goals *modularity* and *exchangeability* are incorporated into the abstract design of MiniWorld. The remaining design goals (Transparency and Performance) depend mostly on the implementation of the *Virtual Network* which is maintained by the *Network Backend* and the *Virtualization Layer*

4.3 Workflow

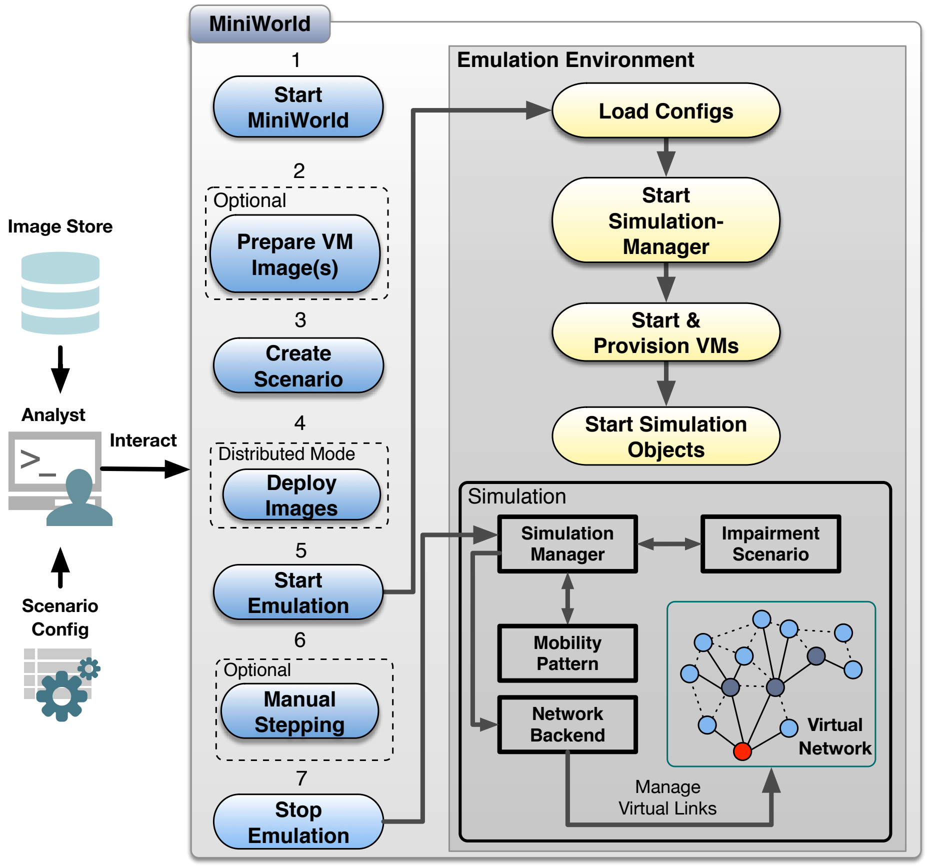


Figure 4.2: MiniWorld Workflow

Figure 4.2 points out the workflow of MiniWorld. Initially, the system has to be started (1). The analyst can create a new VM image or use an existing image from the *Image Store* (2). In this step the software or OS which shall be tested, has to be prepared to be usable with MiniWorld. This requires only a few modifications of the VM and is described in Section 5.7. In the next step a scenario has to be created (3). It bundles the settings of the emulation

such as the number of nodes, the node images, defines the *Network Backend*, *Mobility Pattern*, *Impairment Scenario* etc.
If MiniWorld is used in the distributed mode, the VM images have to be deployed to all computer in the MiniWorld cluster (4).
Finally the emulation can be started by supplying the **Scenario Config** to the MiniWorld interface (5).
The simulation start process of MiniWorld (right side of the Figure) is as follows: First the *Scenario Config* and the **Global Config** are read. The first bundles the scenario description in one config file while the latter holds configuration settings which endure a scenario. These settings are accessible by all objects in MiniWorld. Afterwards the *SimulationManager* and the VMs are started. The *SimulationManager* takes care of the simulation lifecycle such as starting, resetting the simulation and updating the network topology.
MiniWorld needs to know when a VM has finished booting. This is accomplished by either letting MiniWorld know for which string it has to wait (boot mode: *Boot Prompt*) or by simulating pressing enter and waiting for the shell prompt (boot mode: *Shell Prompt*).
Furthermore, nodes are started in parallel to achieve better performance. After all nodes have been started, the VMs are provisioned according to the shell commands supplied in the *Scenario Config*. The mechanism can be used to setup and/or customize nodes.
MiniWorld can be set up to switch the network topology in configurable time steps (normally one second) automatically. Note that an analyst can also manually switch the network topology (6).
The *SimulationManager* cooperates with the *MovementDirector* which keeps track of nodes mobility. It provides a distance matrix which serves as input for the **Impairment Scenario** which governs over node connectivity and link quality. Finally, the *NetworkBackend* is responsible to create or switch the network topology according to the outcomes of the *Link Quality Model* and the *Mobility Pattern*. Furthermore, a scenario can be stopped and afterwards started again (7).

4.4 Scenario Config

MiniWorld aims to be flexible without neglecting the ease of use. Everything in MiniWorld depends on 2 configs: The *Global Config* for settings which are meant to endure a scenario and the description of a scenario with the help of the *Scenario Config*. Both configs are powered by the same config system. For simplicity, both configs are accessible via a singleton class inside MiniWorld.
Configuration of complex scenarios results in long config files which is prone to errors. Hence, the config system allows to limit the values for a specific key via a white-list approach. Moreover, default values reduce the size of configuration files. Mandatory keys can be marked such that an error occurs if no value is defined for a particular key.
A key requirement for the config system is that the change of keys inside the config does not require changes of the config API which is exposed to MiniWorld objects. Hence, a binding between between entries in the config file and the config API is required.

4.5 Interfaces

Virtualized nodes can have multiple interfaces to create separate network segments. Built-in interfaces are *Ap*, *Ad-Hoc*, *Mesh*, *Bluetooth* and *WiFiDirect*. Each node can have multiple

instances of an interface type. Interface instances of the same type are enumerated. An **InterfaceFilter** decides which interfaces are connected to each other. The default *InterfaceFilter* allows only interfaces of the same type and index to be interconnected. A *NetworkBackend* can define its own *InterfaceFilter* to change this behaviour.
Interface types can be treated specially by the *Address Configuration* component. For example they can be put into different network subnets. Future versions of MiniWorld may allow the *Impairment Scenario* to govern over link quality based on distance and additionally the interface type. *Bluetooth* connections for example may have a lower bandwidth than *Mesh* connections.
There are 2 special interfaces: The first is the **Hub** interface which connects all interfaces to one or more hubs. These hubs are internally represented by a **CentralNode**. It is up to the network backend to provide a class for this purpose.The *Hub* interface models a scenario, where all nodes are in the same collision domain without mobility involved. It provides an easy opportunity to simulate scenarios such as a football game where people are crowded. Only static link impairment can be applied since there is no mobility.
The other special interface is used for management purposes. In contrast to the *Hub* interface, a **Management** interface is not affected by link quality impairments. It serves as a management/side channel which can be used in experiments for control information or ssh automation. A *ManagementNode* has to be provided by a *NetworkBackend* to support the management network.

4.6 NetworkBackend Communication

The core of the framework handles starting the virtual nodes and setting up the network. In the following the emulation start process is explained. Afterwards the integration of a network backend into the MiniWorld emulation framework is depicted.

4.6.1 Simulation Lifecycle

Figure 4.3 illustrates the emulation start process which is triggered by the user via an IPC interface. The *SimulationManager* coordinates the start process. First a lock is acquired to prevent multiple starts of a scenario. Moreover, the lock blocks an IPC call until all nodes have been started and the first network topology has been created, but only if the *RunLoop* is used. After the first step of the *RunLoop*, the lock is released.
The **NetworkBackendBootstrapper** object holds the types of the emulation node, the network backend, the switch type, the connection type etc. It servers as a container for all types which are exchangeable in MiniWorld. In the **NetworkBackends** module, the bootstrapper object is dynamically populated with the types, especially the network backend what shall be used.
After the bootstrapper has been created, the network backend type is taken from the bootstrapper and created. Note that the bootstrapper object also solves recursive import problems between for example the network backend and a connection object. Static attributes and the class can be accessed via the bootstrapper. Furthermore, the network backend can be used everywhere in the code because it is available as a singleton.

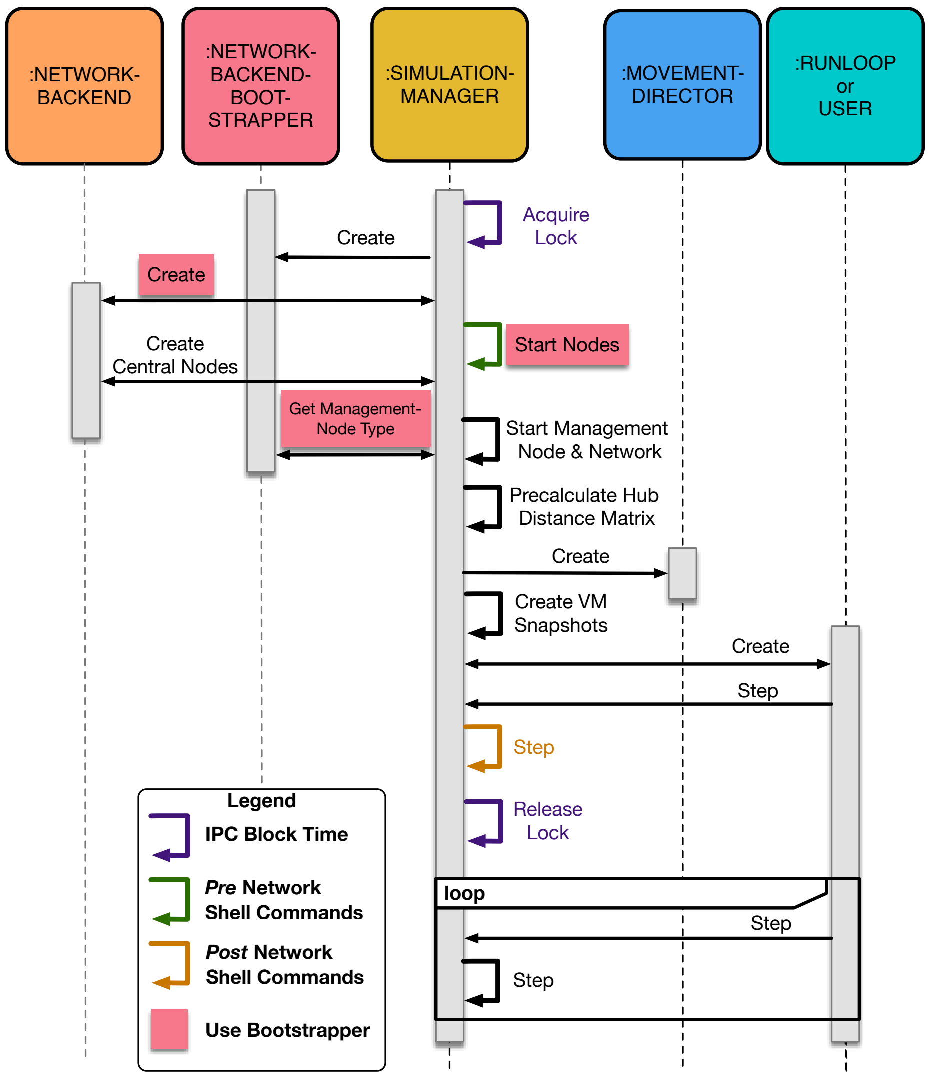


Figure 4.3: Simulation Lifetime Sequence Diagram

Afterwards the nodes which are provided by virtualization are started in parallel[1]. Node provisioning allows shell commands to be executed before and after the first network topology has been started. **Pre Network Shell Commands** are executed on the VMs after they have been started, hence they are not part of a VM snapshot.
The *Management Network* is set up directly after all nodes have been started, but after the *Pre Network Shell Commands* have been executed.
The hub interface has to be represented in the distance matrix since static link quality impairment has to be applied (based on the distance). The *CentralNodes* have an own id which needs to be respected in the distance matrix. Therefore for each connection to the *CentralHubs*, an entry with WiFi distance is generated. The pre calculated distance matrix for the *CentralHubs* is merged with the distance matrix from the *MovementDirector* afterwards and improves performance due the caching.
Finally, the *MovementDirector* is created and snapshots of the VMs are taken. The next start of the same scenario leverages the snapshots to improve overall start times.

[1]Depending on the settings of the *Scenario Config*

Stepping can be done either from the *RunLoop* which calls the step method in well defined time intervals (default: one second) or manually from the user. In both cases, the network topology is created according to the distance matrix and the *Impairment Scenario*. Finally, the lock is released. Note that the lock is released even further if no *RunLoop* is used. Moreover, the *Post Network Shell Commands* are executed in parallel.
The rest of the emulation is controlled by the stepping of the *RunLoop* and illustrated in the following.

4.6.2 Simulation Step

A well defined interface[2] describes the control flow between the *SimulationManager*, the *NetworkManager* and the *NetworkBackend*. Figure 4.3 outlines the integration of a network backend in MiniWorld. The *NetworkManager* uses the information exchange between the *SimulationManager* and the *NetworkBackend* to keep track of currently established connections and the connection link quality (**Connection Tracking**, blue line on the *SimulationManager* axis). This enables the *SimulationManager* to provide the *NetworkBackend* with different callback methods to configure the virtual network.
The different callbacks are discussed in the following. There are different control flows depicted in the Figure depending on the return types of the callback methods. Callbacks which are not supposed to return anything, are marked with black arrows. Callbacks with a return value are marked red. Code which is executed in the object itself is depicted with a blue arrow.
Connection_across_servers is called for 2 nodes where at least one node is not represented locally by means of *virtualization*. The method can be leveraged by a *NetworkBackend* in the distributed mode to create tunnels. For that purpose the IP address of the remote server, which is hosting the remote node, is supplied.
Some of the callbacks provide a method before and after a certain action has been performed. For example there are two callbacks for the *simulation_step* prefixed with either *before* or *after* and are executed at the start and end of the whole step lifecycle.
Changes in the distance matrix are propagated to the *NetworkBackend* with both the full distance matrix as well as the changes between two distance matrices. This notification is used by the *NetworkManager* to inform the EventSystem about the number of connections that have to be established or taken down. This is necessary to correctly display the simulation progress.
In the **before_link_initial_start** callback, the *NetworkBackend* is expected to create a switch and a connection for a new link between two peers. It is up to the *NetworkBackend* to reject the creation of a new link. This is used by the *NetworkBackend* presented in Section 4.10.2.
The aforementioned method deals with the creation of new links. Links are viewed by MiniWorld as stable. Therefore **link_up** and **link_down** are only called for existing links. If the distance changes for two nodes, the *LinkQualityModel* decides whether the link shall be taken down or up. Note that the callbacks are only notified if the links status really changes. If the distance changes but the links status does not, there still might be a change in the link quality. The **before_link_quality_adjustment** callback gives the *NetworkBackend* the opportunity to change the link quality.
Finally, depending on the *IP Provisioner*, the IP addresses of interfaces inside the node are changed. Additionally, connectivity is checked by the *NetworkManager* on IP basis.

[2]NetworkBackendNotifications

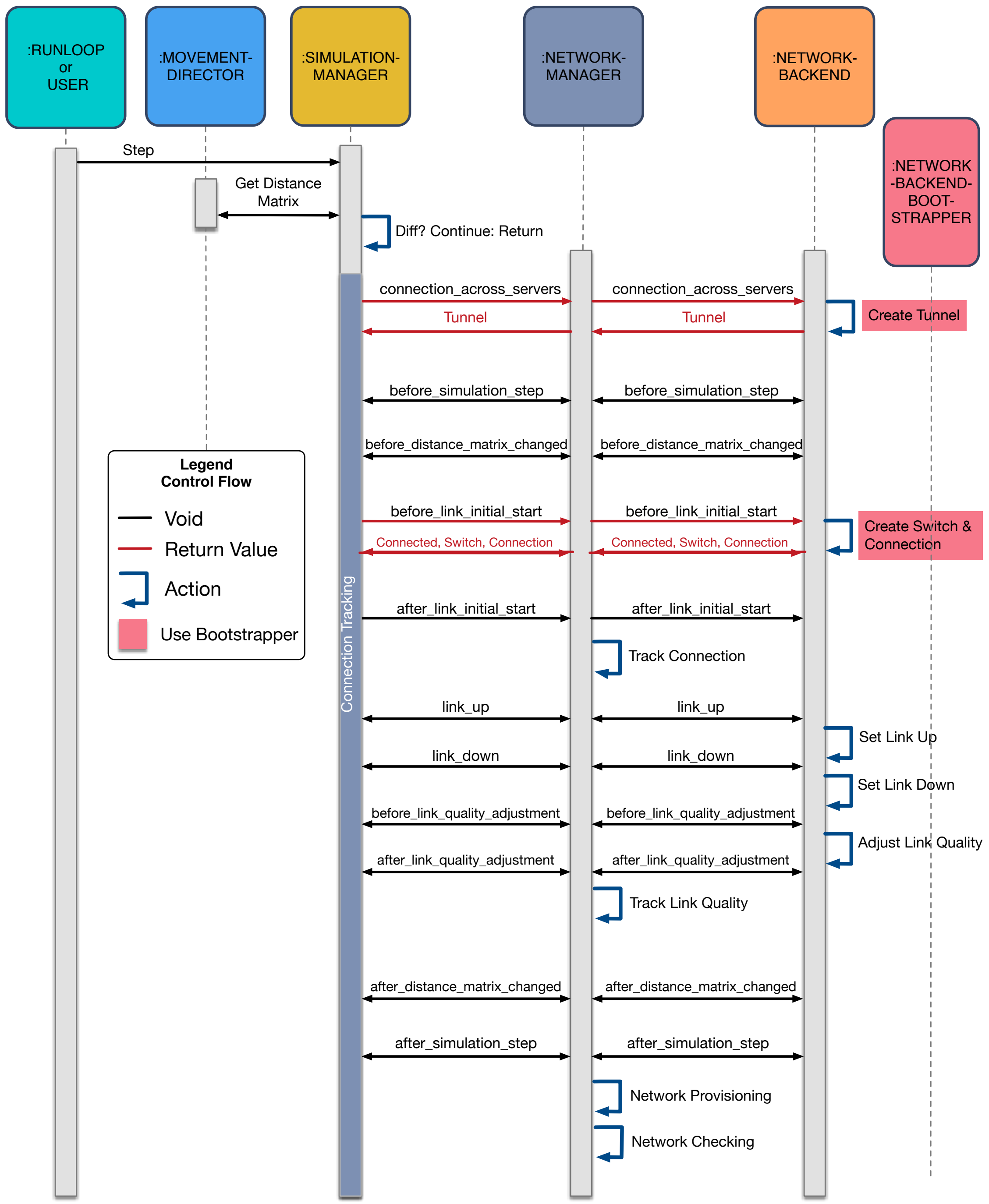


Figure 4.4: Step Sequence Diagram

4.7 REPLable

There are many external processes with whom MiniWorld needs to communicate. The interface offered by whose processes is mostly text based. Input is sent to processes and output returned. Since there is only text, no return codes or exceptions are available. Thus, the output has to be parsed according to specific patterns. There are two approaches to

accomplish that: Either the successful output for some input is known, or there is pattern in the output which points out an error.
REPL is the acronym for read-evaluate-print-loop. There are several components that are expected to make use of the *REPLable* mechanism. One of them is the virtualization layer since most virtualization software is capable of exposing the serial console of the VMs via IPC. With the help of the *REPLable* mechanism it is possible to get a shell to a VM without relying on tools such as ssh which requires network access.
Unfortunately, a serial console which is exposed via IPC does not offer the same functionality as a real shell does since only text is returned. Hence, there is not such fine grained control over the process compared to a real shell. MiniWorld needs to know then the process is finished and which output belongs to sent input. Therefore, each class leveraging the *REPLable* mechanism, needs to define a **Shell Prompt**. This prompt is printed to stdout by a shell when a process exits or the user presses enter.
The *REPLable* mechanism is depicted in Figure 4.5. There are 3 classes illustrated. The first object derives from the *REPLable* interface. It offers services that rely on external processes. Additionally, it can check the return code of the executed commands. Note that there is no blocking connection[3] to the IPC service. Therefore, for a list of commands to be executed on the VM, a new connection to the interface has to be created. The Figure outlines that the connection establishment process blocks until the IPC interface is reachable. Some services may not be ready or need the enter key to be pressed. This is simulated by sending a $\backslash n$. The processes is called *Shell Entering*. Newlines are sent until the *Shell Prompt* has been read. Each class is connected to a *TemplateEngine* which allows variables to be replaced by strings. The variable {node_id} for example gets replaced by the actual node id. Commands are postfixed with a newline and sent to the IPC interface. Data is read until the defined *Shell Prompt* has been read. Note that multiple commands can be sent at once to the IPC interface, thus improving performance. A class implementing the *REPLable* interface has the possibility to implement additional checking for failures. This can be leveraged to check the return codes of executed commands.

4.8 Link Quality Models

A *LinkQualityModel* controls the impairment which is applied to the virtual network (*Impairment Scenario*). Static impairments are supplied via the constructor to the *LinkQualityModel* and serve as default values. For each step and distance between two nodes, the *SimulationManager* calls the *LinkQualityModel*. First, the model controls if a connection shall be established at all. Second, the link quality is determined. For that purpose, the distance can be taken into account.
The link quality is described by a dictionary. Not all network backends have the same possibilities to impose link quality. Therefore, it is not possible to define a common format of link quality models. Instead, it is up to the *NetworkBackend* to understand the keys and values used by a *Link Quality Model*.
Currently there are 3 simple models shipped with MiniWorld. The first is a **Fixed-Range Model** where nodes are interconnected if their distance is less than 30 meters. The bandwidth is stable during a scenario and taken from the *Scenario Config*. Note that this model is also usable for LAN emulation, where no mobility is involved. A *MovementPattern* using the fixed-range model simply has to define the range between two peers between 0 and 30.

[3]The IPC interface is held open for shell access by the user

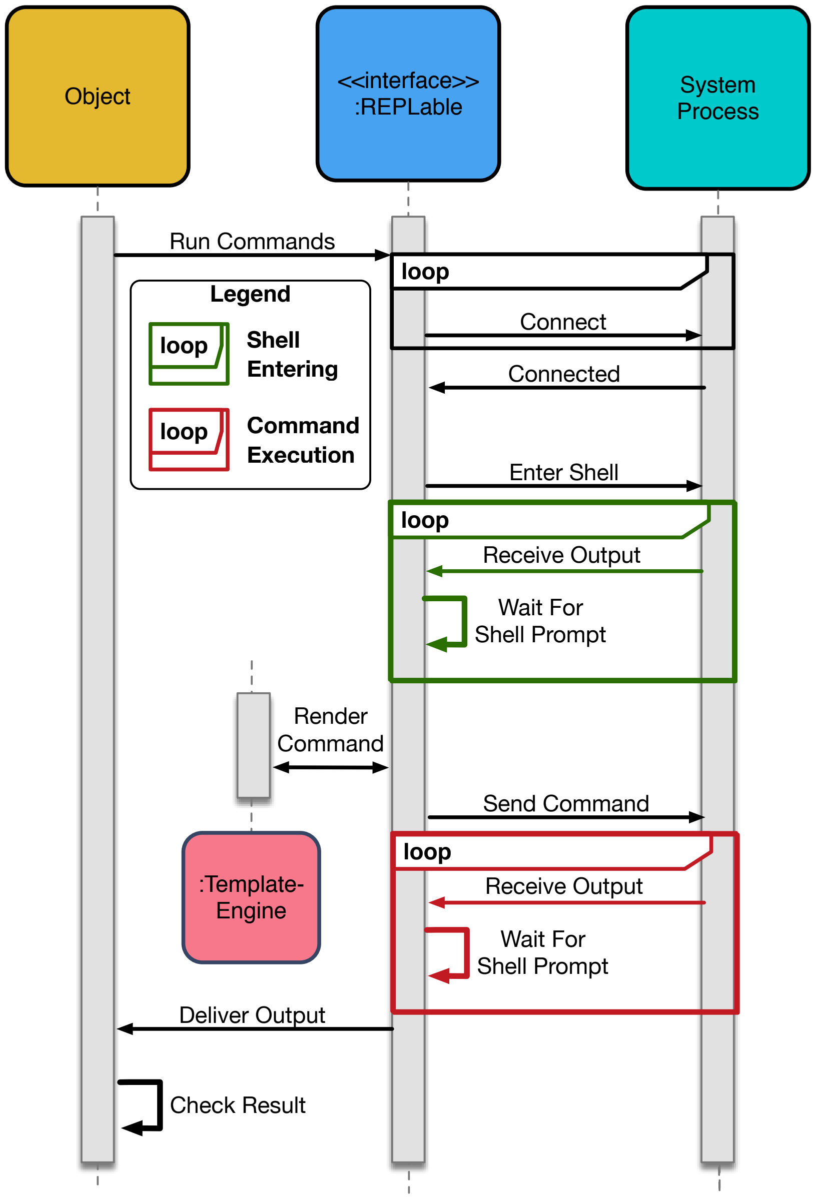


Figure 4.5: Sequence REPLable

The second model (**WiFi Simple Linear**) varies bandwidth and delay with respect to the distance. It needs a maximum bandwidth to be defined. If no bandwidth is defined in the *Scenario Config*, 54 Mbps/s is assumed. The bandwidth decreases linear and the delay increases linear with the distance. The third model (**WiFi Simple Exponential**) halves bandwidth and doubles delay every four meters.
Both WiFi models are limited to the *Bridged Network Backend*.
Table 4.1 illustrates the 2 simple WiFi models for distances between 0 and 29. For distance 0 and 1 the values stay the same. Beginning with distance 2, the bandwidth decreases linear with the distance in the *WiFi Simple Linear* model. The *WiFi Simple Exponential* model decreases bandwidth every four meters.
Note that both WiFi models define a variable delay which is used in addition to the constant delay. Therefore the delay is Delay Const $\pm$ Delay Var where the delay depends to 25% on

	WiFi Simple Linear Model			WiFi Simple Exponential Model		
Distance	**Bandwidth**	**Delay Const**	**Delay Var**	**Bandwidth**	**Delay Const**	**Delay Var**
0	54000	0	0.0	54000	1	0.1
1	54000	0	0.0	54000	1	0.1
2	36000	1.0	0.1	54000	1	0.1
3	27000	2.0	0.2	54000	1	0.1
4	21600	3.0	0.3	27000	2	0.2
5	18000	4.0	0.4	27000	2	0.2
6	15429	5.0	0.5	27000	2	0.2
7	13500	6.0	0.6	27000	2	0.2
8	12000	7.0	0.7	13500	4	0.4
9	10800	8.0	0.8	13500	4	0.4
10	9818	9.0	0.9	13500	4	0.4
11	9000	10.0	1.0	13500	4	0.4
12	8308	11.0	1.1	6750	8	0.8
13	7714	12.0	1.2	6750	8	0.8
14	7200	13.0	1.3	6750	8	0.8
15	6750	14.0	1.4	6750	8	0.8
16	6353	15.0	1.5	3375	16	1.6
17	6000	16.0	1.6	3375	16	1.6
18	5684	17.0	1.7	3375	16	1.6
19	5400	18.0	1.8	3375	16	1.6
20	5143	19.0	1.9	1688	32	3.2
21	4909	20.0	2.0	1688	32	3.2
22	4696	21.0	2.1	1688	32	3.2
23	4500	22.0	2.2	1688	32	3.2
24	4320	23.0	2.3	844	64	6.4
25	4154	24.0	2.4	844	64	6.4
26	4000	25.0	2.5	844	64	6.4
27	3857	26.0	2.6	844	64	6.4
28	3724	27.0	2.7	422	128	12.8
29	3600	28.0	2.8	422	128	12.8

Table 4.1: Link Quality Models WiFi: (a) Linear Model (b) Exponential Model

the last one.
LinkQualityModels may become very complex. To prevent them from slowing down the connection switching process, the link quality settings for distances between 0 and 100 are precalculated (**LinkQuality Caching**). For that purpose, distances are rounded. This approach is a tradeoff between performance and granularity.

4.9 Movement Patterns

In the following a short overview of the *Mobility Patterns* is provided. Note that most of them have been implemented by Patrick Lampe who contributed code to MiniWorld during an internship.
There are 4 different mobility patterns implemented at the time of writing. The first 2 are based on the Open Street Map (OSM)[4] of Marburg. With the help of the OSM map, a **RandomWalk** and a **MoveOnBigStreets** pattern is implemented. The second prioritizes big streets where possible. The third *Mobility Pattern* is based on the **Arma 3** game[5] where coordinates are extracted from the game for each player to feed a *MovementDirector* with node positions.
The last *Mobility Pattern* is the **CORE Mobility** pattern. It leverages the UI of the CORE emulation platform to place nodes graphically. Each topology file can be exported to XML. The basic idea of the *CORE Mobility* pattern is to have multiple topology files. Each topology can be switched after a predefined number of time steps. Moreover, there are different parsers for the topology files. In the **LAN** mode, only the connections are extracted from the topology file. For this mode, wired connections have to be used inside the UI. A *Fixed-Range Model* inside CORE decides about the connectivity of nodes which is then parsed. In the **WiFi** mode, wireless connections have to be used. Instead of checking which node are interconnected, the coordinates of the nodes are parsed. The *LinkQualityModel* of MiniWorld then decides about connectivity and link quality. Note that the canvas where nodes are drawn onto should be resized so that nodes node distanced can be estimated in the UI.
The *CORE Mobility* pattern has been used for testing and also serves as a basis for the experiments which are presented in Section 6. For most applications, repeatability is very important. Together with the ability to define how the topology changes make this *Mobility Pattern* very useful.

4.10 Network Backends

A network backend is one of the fundamental components of MiniWorld. It creates the virtual network and takes care of adding and removing connections as well as adjusting the link quality.
Network backends are supposed to follow an object-oriented design (but are not enforced) where switches, links and tunnels are modeled as a class. To support the special interfaces **Management** and **Hub**, they can provide special virtual nodes classes which take care of starting and configuring the node(s). These are called *ManagementNode* and *CentralNode* respectively. In addition, it may support distributed virtual networks. These management, hub and distributed links may require custom setup but are integrated into the normal

[4]`http://www.openstreetmap.org`
[5]`https://arma3.com`

NetworkBackendNotifications between the *SimulationManager* and the *NetworkBackend*. The **ConnectionInfo** is supplied as parameter in methods of the **NetworkBackendNotifications** interface and can be used to distinct between code for the management, hub or the distributed network.
Some network backends may also require custom command line arguments or special behaviour of the virtualization layer. Therefore, a custom **VirtualizationLayer** class can be used. The types of these custom classes are defined in the **NetworkBackendBootStrapper** object. The custom *VirtualizationLayer* class and the *ManagementNode* are created by the *SimulationManager*, the others by the network backend itself.
An *InterfaceFilter* gives a network backend the control which interfaces can be connected to each other but the *SimulationManager* still enforces that only interfaces of the same type can be connected.
Moreover, each network backend can influence if a custom *Network Configurator* is needed. The *Bridged Lan* network backend for example requires a special configurator for point to point links.
MiniWorld comes with 4 different network backends: The first is based on VDE (see Section 3.2.2). The second and third make use of Linux bridges. The fourth is a fork of *wmediumd* and based on the *mac80211_hwsim* wifi simulator (See Section 3.3.3).
The **VDE** network backend is presented in Section 4.10.1. The **Bridged** based network backends are depicted in Section 4.10.2. Finally the **WiFi** network backend which enables real wireless devices inside the VMs is illustrated in Section 4.10.3.

4.10.1 VDE

Virtual Distributed Ethernet has already been introduced in Section 3.2.2. It comes with a few user-space programs: The most important ones are *VDESwitch* (a switch) and *Wirefilter* (link/cable between switches). Both provide an interface via a UDS socket. Therefore, MiniWorld leverages the *REPLable* component to enable communication with the VDE components. The **VDESwitch** interface allows to set the hub mode, manage VLAN, viewing switch ports and their links, enable color mode, set the number of ports and to enable the fast spanning tree protocol. The **Wirefilter** interface allows to define the link quality in terms of loss, delay, duplicate packets, bandwidth, speed, noise, MTU and even more advanced modes which make use of markov chains.
MiniWorld provides Python wrappers for both objects, where calls are transparently redirected to the UDS socket. Both check the return codes of executed commands by checking for specific text patterns.
Figure 4.6 points out the architecture of the VDE network backend. 2 VMs are illustrated in the Figure: One on the top and one on the bottom. The VMs have several interfaces where each is connected to a *VDESwitch*. This enables to apply different link qualities for different interface types. Note that a node can have more than one interface of the same type. Each interface type is additionally put into a VLAN. In the Figure *n* represents a variable amount of interfaces. Moreover, the special *management* interface is shown. A normal interface is connected to another normal interface with a *Wirefilter* between them. This design allows for each connection between two peers to define a custom link quality and emulates WiFi behaviour where nodes have different signal strengths depending on the distance and obstacles between them etc.
The color patch introduced in Section 3.2.2 allows the creation of a hop-2-hop network instead of a single collision domain where all peers can see each other. Traffic is only

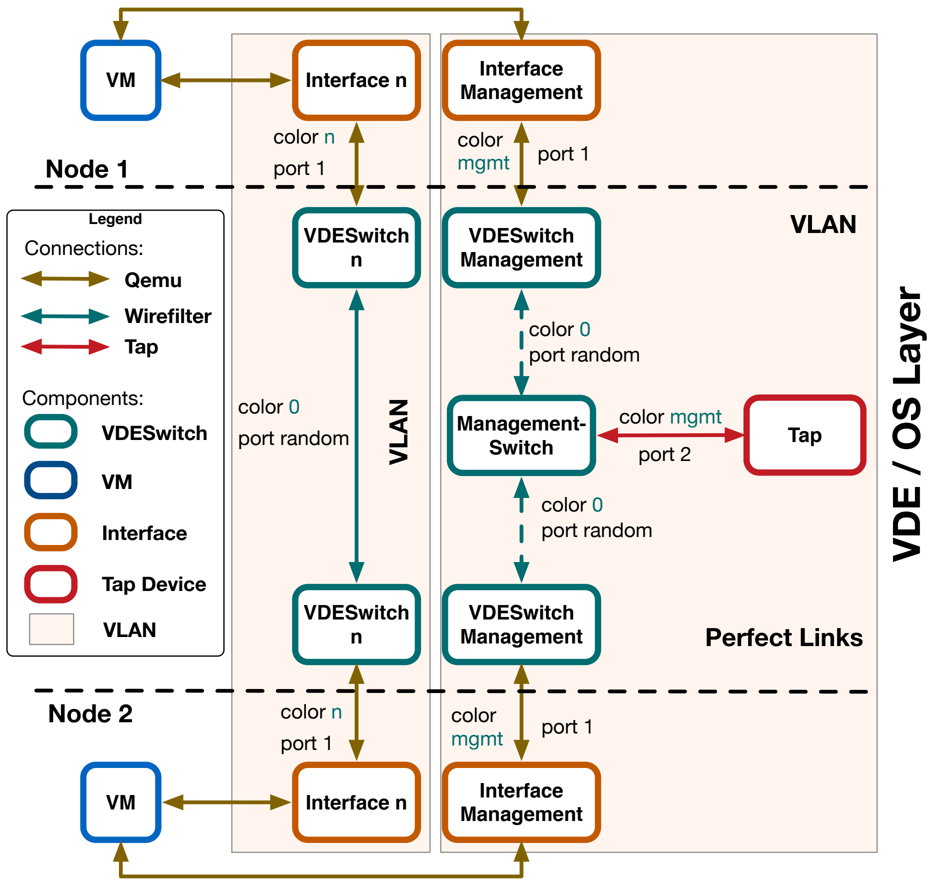


Figure 4.6: VDE Network Backend

forwarded between switch ports if their color differs. Note that a *Wirefilter* between two *VDESwitches* always has color 0 by default. The interfaces of the VMs are connected to the appropriate switches at boot time with a special command line parameter for the *VirtualizationLayer*. These connections have a color which is $\neq 0$. They are colored with the index of the interface type which is common for the same interface type. Therefore, if 2 switches are interconnected with a cable (*Wirefilter*), they can communicate because color n is not equal to 0. Imagine, node 2 is connected to a third node. Node 1 cannot directly communicate with node 3, because the *Wirefilter* has color 0 and packets are dropped if the color of the incoming and outgoing port are the same. Communication between node 1 and 3 is not directly possible and requires routing of node 2. This is the scenario of a WMN.
The **Hub** interface is also implemented for the *VDE* backend, and works as illustrated by *Interface n*: Each node's switch has a connection to any other.
The management network is quite different. There is one or more switches which are not tied to a VM directly but are connected to a tap device on the host (*CentralNode*). Each node's switch is connected with a *Wirefilter* to one of the *Central Switches* which are also interconnected. Therefore, node communication is not allowed. Instead, only the host can communicate with the nodes and vice versa. The management network provides a perfect channel instead of the other interfaces where link quality is applied by the *Wirefilters*.
Moreover, all calls to the UDS sockets are written to stdout. Depending on the logging level, the UDS outputs are also shown.

All switches except the one used for the management network, is set to operate as hub, mimicking the wireless broadcast nature.

4.10.2 Bridged

The previous network backend used components running in user-space. In the following the *Bridged Network Backend* is introduced to the reader. It leverages technologies from the Linux kernel to create the virtual network. The most important parts are Linux **bridges** and Linux **TC**. There are 2 *Bridged Backends* which differ in the way connections between two peers are handled. The **Bridged LAN Network Backend** multiplexes connections via a single tap interface per MiniWorld interface, connected to the **Virtualization-Layer! (VirtualizationLayer!)** process. In contrast, the **Bridged WiFi Network Backend** uses one interface to represent a connection. For that reason, the latter backend is viewed as static because the number of unique connections has to be known beforehand to set up the number of NICs in the VMs. The first backend is considered as dynamic because it does not matter how many unique connections are going to exist.
Due to the static and dynamic characteristics of the *Bridged Network Backends*, the *Bridged LAN Network Backend* is a good candidate for emulating wired networks, whereas the *Bridged WiFi Network Backend* fits to the demands of wireless networks.
Figure 4.7 outlines the architecture of the the *Bridged Network Backend*. Both nodes (top and

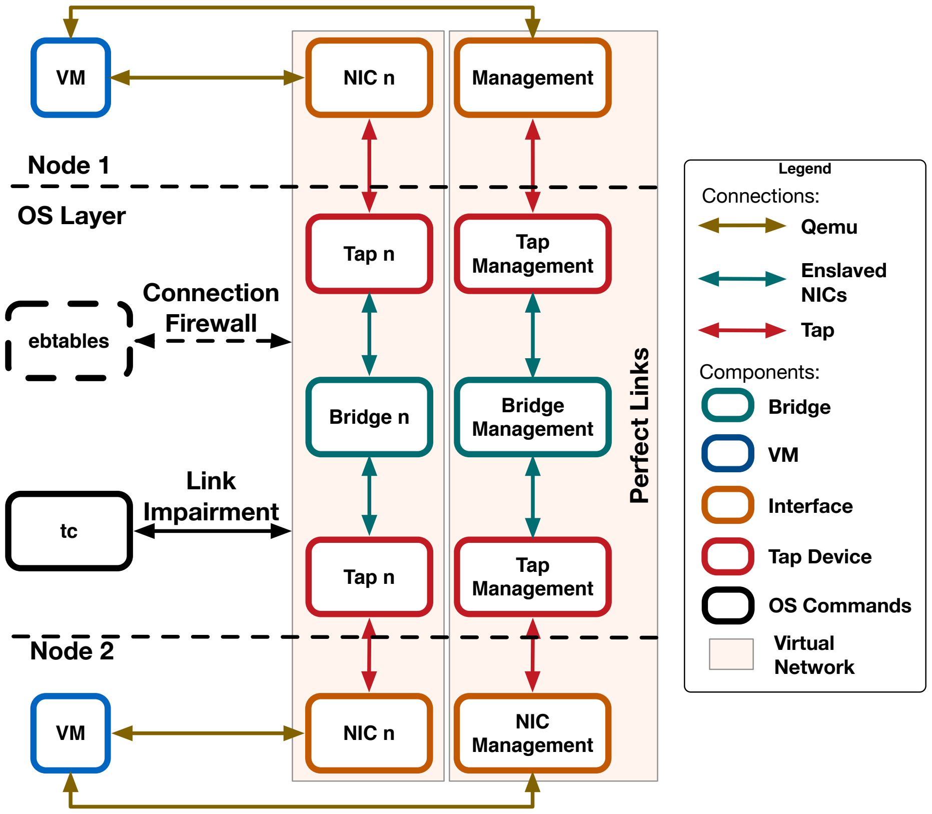


Figure 4.7: Bridged Network Backends

bottom of the Figure) have separate interfaces which are provided by Qemu in terms of a *Device Model* to the VMs. Each NIC (orange box) is associated with a tap device (red box) on

the host. The user-space part of the tap device is a socket managed by Qemu. The creation of the virtual network is done on the host, hence no involvement of the guests is required. Tap devices are added to bridges. Because a NIC can only be enslaved to one bridge at the same time, either multiple NICs are necessary inside the guests or connections have to be multiplexed. These aforementioned differences make up the two different *Bridged Network Backends* and are discussed in the following.
Both network backends rely on shell commands for the virtual network creation instead of UDS automation used by *VDE* network backend. For that purpose, the *ShellCommandSerializer* is used.

Bridged LAN

The *Bridged LAN* network backend is theoretically not limited to static topologies. Rather the number of connections have to be known beforehand. The *CORE Mobility Pattern* allows to define a collection of topologies together with the number of steps they shall remain.
The *Bridged LAN* network backend determines for each node the maximum number of connections during the whole scenario. The VMs are then created with the necessary number of NICs. Therefore, the *Bridged LAN* network backend is able to emulate WiFi networks because each connection is represented by a NIC and thus to each connections impairments can be applied with the Linux traffic shaping facilities. The two tap devices which are part of a connection are put on a single bridge.
Note that the *CORE Mobility Pattern* is the only one at the time of writing which is supported for this network backend. For other mobility patterns the distance matrix and thus also the connection matrix is not known beforehand.
The representation of a connection with a single NIC basically creates point to point links. Hence, two NICs which model a connection, are put into the same subnet[6]. The network backend aims at being dynamically even if it is usable only for static connections. An idea to overcome the limitations that the number of NICs need to be known before the VM is started, is the use of hotplugging new NICs on demand by using the Qemu monitor connection by means of UDS automation. Because this might be implemented in newer versions, the network backend is kept dynamically. Therefore, an IP address is allocated to the guest NIC first, then a connection is established.
The *Bridged LAN* network backends *InterfaceFilter* allows all devices to be connected to each other. Therefore, the *before_link_initial_start* callback method is triggered for every tap device combination of two nodes for a new connection. Internally, the network backend remembers which tap device is used and which is not. The network backend may reject a connection in this callback when a tap device is already in use. The *SimulationManager* then skips the *link_up* and *link_down* callbacks for the two tap devices. Moreover, the *NetworkManager* notices that no connection has been established, therefore no other callback methods such as for the link quality adjustment are called.

Bridged WiFi

In contrast to the aforementioned *Bridged Network*, the *Bridged WiFi* backend requires only a single NIC to represent multiple connections. Therefore, for each MiniWorld interface a NIC is configured for the appropriate VM.
Ebtables is the equivalent of *iptables* to create firewalls, but operates on the link-level instead.

[6]This subnet size and the IP ranges are configurable in the *Scenario Config* with the CIDR notation

Then a new connection shall be established, *ebtable* rules based on the tap device names are used for filtering. Additionally, connections are tracked in the Linux kernel by marking them with an integer number. This enables the Linux traffic control facilities to apply different link qualities based on the marked connections. Note that the aforementioned network backend simply used the tap device to identify a connection.
The IP provisioning process is easier because each NIC can be configured once. Each interface type is put on the same subnet where IP addresses are allocated starting by a predefined CIDR IP range.

Execution Modes

There are different execution modes which vary in the way the virtual network is setup. In the first execution mode the network is setup with *iproute2* commands only. *iproute2* requires an up to date version compiled for the current kernel to support bridge commands. The execution mode *iproute2* provides high performance but requires custom setup. Therefore, the execution mode *brctl* replaces all *iproute2* bridge commands with the appropriate *brctl* command. Both execution modes rely on shell commands. A third execution mode uses *pyroute2*[7]. It is a python library which communicates directly with the kernel via a netlink socket and can be seen as the *iproute2* equivalent for python.
In addition to the execution modes, there are different ways to execute shell commands. Commands can be executed either sequentially or **Parallel**. Some commands even support **Batch** operations which reduces internal overhead. The third option combines multiple commands in one shell command (**One Shell Call**). Section 6.4.4 provides benchmarks for the different execution modes and options.

4.10.3 WiFi

The *WiFi Network Backend* is not a pure network backend in MiniWorld. Instead it is a combination of a real network backend with a user-space process inside the VM. Figure 4.8 illustrates the design of the *WiFi Network Backend*. The heart of the backend is the fork of *wmediumd* user-space program (light blue) and the *mac80211_hwsim* radio simulator which provides the *WiFi Nic* to the Virtual Machine (lightgray arrows).
Wmediumd uses the *nl80211* interface to get the 802.11 frames from *mac80211_hwsim*. The program emulates the 802.11 MAC layer with exponential backoff, QoS and signal strength. Afterward these effects have been applied to the frames, they are send back to the kernel.
MiniWorld leverages the program as the basis for the *WiFi Network Backend* but drops the wireless medium emulation, because it is very slow. Instead, frames are retrieved from the wireless NICs and encapsulated into UDP datagrams which are send to a multicast group. In the guest, multicast has to be routed via one of the devices which is provided by some of the real network backends. At the receiving side, the 802.11 frames are extracted from the UDP datagrams. Note that currently no impairment functions are available for the wireless network backend.
The backend provides real WiFi device inside the guest which can be used to test the real *mac80211* and/or *open80211s* layer inside the Linux kernel. An example *Scenario Config* is presented in Section 35. Note that the user-space program needs to be run inside the guest, hence the approach is limited to Linux.

[7]`https://github.com/svinota/pyroute2`

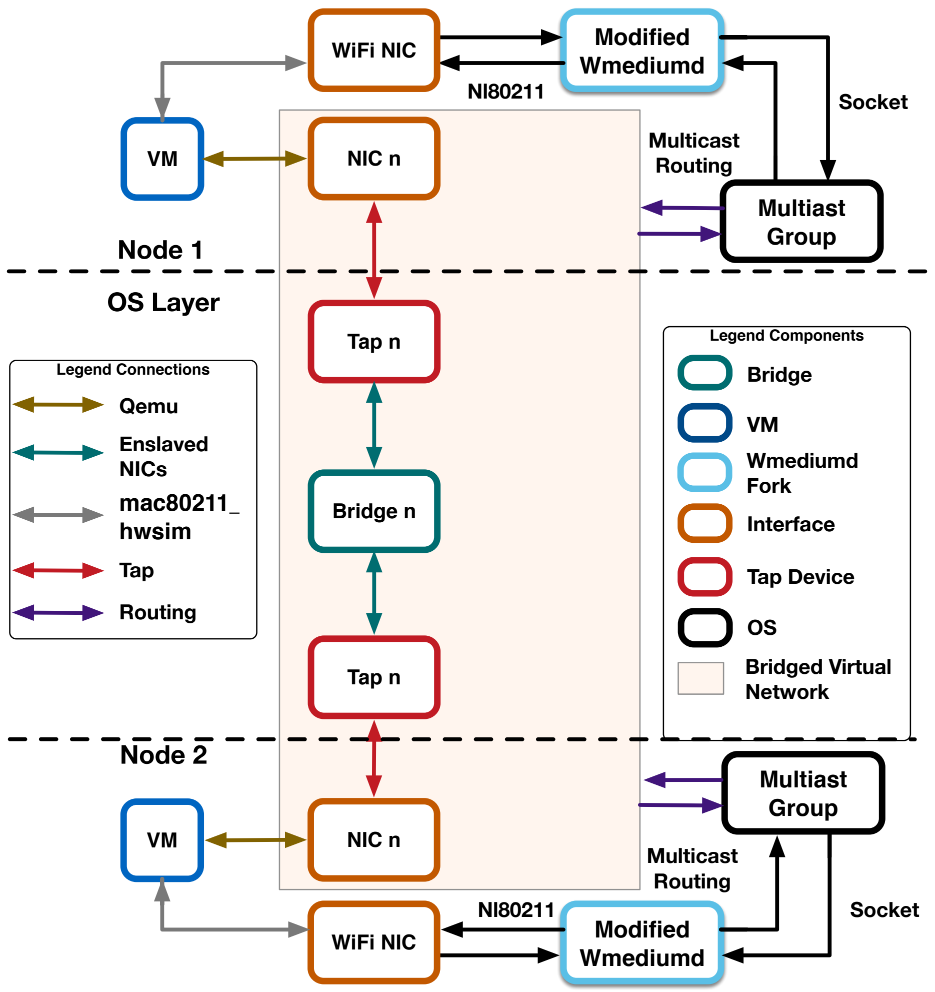


Figure 4.8: WiFi Network Backend

4.10.4 Summary

The integration of 4 network backends into MiniWorld has been presented. The **Bridged LAN** network backend is not suited to emulate wireless network, since the wireless broadcast nature is not taken into account. The **VDE**, **Bridged WiFi** and **WiFi** network backend mimic a broadcast channel, hence nodes which are interconnected, receive frames also from its neighbours. This is achieved by the *VDE* and *Bridged WiFi* network backends by enabling the hub mode of the switches. The *WiFi* network backend uses a multicast group, therefore frames are distributed to all nodes.
The *WiFi* network backend does not offer link impairments at the time of writing whereas the *Bridged* network backends leverage the Linux TC system. The *VDE* network backend uses *Wirefilters* to model the link between nodes.

4.11 Distributed Mode

The distributed mode of MiniWorld aims at reducing the bottleneck that appears if a single machine is out of resources. This may be due to missing CPU or memory resources. The distributed mode allows each Linux computer to take part in the emulation. In the following, the node scheduling is introduced to the reader. Afterwards the distributed architecture is presented.

4.11.1 VM Scheduling

The placement of virtual nodes on computers is essential for the distributed mode. The resources of the computers have to be taken into account, to improve the scheduling decisions. Server scores are communicated by each computer which takes part in the emulation. The score includes the CPU and memory resources because they are the potential bottlenecks. Network bandwidth would be nice to know, but is not part of the score at the time of writing.
During the boot process of a Linux machine, a value is calculated which represents the performance of CPU. The value is called *Bogomips*. Together with the free memory it serves as input for the scheduler.
There are two so called *NodeDistributionStrategies* shipped with MiniWorld. The first is pretty straightforward and assigns each computer the same amount of VMs (**NodeDistribution-Equal**). A better approach is to make use of the score. The **NodeDistributionScore** allocates the VMs according to the bogomips each computer has. Then, for each computer it is check that the amount of free memory is not exceeded. Otherwise, the amount of VMs is reduced until the memory fits the needs of the Virtual Machines. The unscheduled VMs are afterwards distributed among all machines according to the CPU score. Note that individual VM settings are not taken into account yet.

4.11.2 Communication

The distributed communication is build with ZeroMQ (ØMQ). In early prototypes RPyC has been used to coordinate communication among a group of nodes but showed to be very slow for more than round about ten clients.
ZeroMQ: ZeroMQ is a networking library written in C++. The ØMQ Guide [21] explains it as following:"It's sockets on steroids". It is a very good summary, because basically it provides very advanced socket types with various transports such as: in-process, inter-process, TCP and multicast. Many to many or one to one communication can be depicted with a single socket. There are many socket types built-in which help developing concurrent applications.

4.11.3 ZeroMQ Communication Patterns

There are two important pattern used by MiniWorld.
Request-Reply Pattern: The first is the *Request-Reply* pattern where to each request a reply belongs. The req and rep sockets can be used to implement this pattern which mimics the behaviour of RPC. Note that any other order of messages is not allowed by ZeroMQ. The pattern can be extended by Dealer sockets, which advances the pattern for many to one communication. Messages from N clients are faire-queued and provided by a single socket.
Publish-Subscribe Pattern: The other pattern is the *Publish-Subscribe* pattern where multiple clients subscribe to a publisher. It is a form of one to many communication and very fast because communication is only one way. The pattern is implemented with two different

socket types [21, 31].

4.11.4 Distributed Architecture

Figure 4.9 outlines the distributed architecture of MiniWorld. There are two clients and one

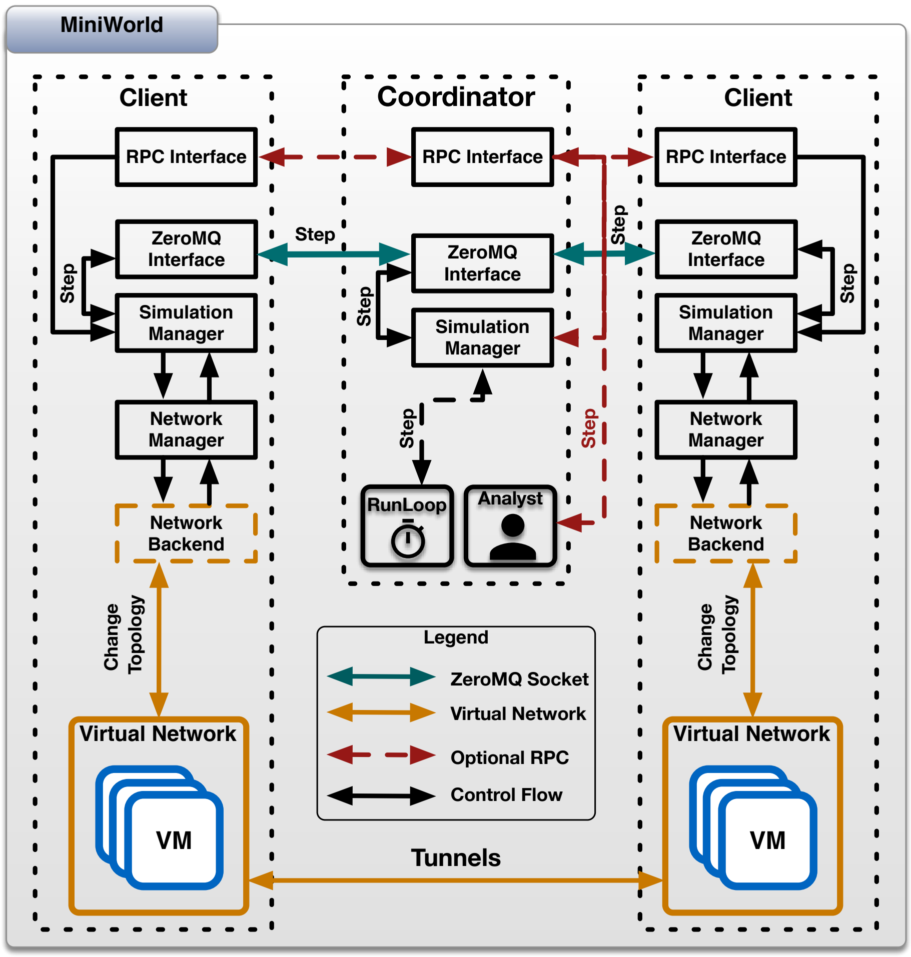


Figure 4.9: MiniWorld Architecture Distributed

coordinator. The user can interact with any of these via the Remote Procedure Call (RPC) interface to query information or execute commands on specific nodes. Note that direct access to the RPC interface of the clients is not necessary. Instead the CLI interface connects to the *Coordinator* RPC interface by default and redirects requests to the appropriate client if necessary. The RPC interface of the *Coordinator* runs on a different port so that the *Coordinator* and *Client* can co-exists on the same machine.
Starting a scenario and manually stepping can be accomplished by user via the CLI interface. Note that the *RunLoop* is located on the *Coordinator* only. A step of either the *RunLoop* or the user is communicated via the ZeroMQ layer for performance reasons. The distance matrix can be distributed either with the request-reply or the publish-subscribe pattern.
In the case of the publish-subscribe pattern the distance matrix is sent to any subscriber and filtered on the client-side so that only the part of the distance matrix is sent to the

SimulationManager which contains nodes for which the client is responsible.
In the case of the request-reply pattern, the distance matrix can optionally be filtered on the coordinator-side because it holds to each client a separate connection in contrast to the publish-subscribe pattern.
The serialization format of messages is either *JSON* or *MessagePack* and configurable as well as the communication pattern via the *Global Config*.
The ZeroMQ layer is modeled as a Deterministic Finite Automaton (DFA). The number of clients is set in the *Global Config* and picked instead of a time based solution where clients can register in a time slot.
The communication is asynchronous. For each state, clients need to send a request to the coordinator. First if all requests arrived, the coordinator replies. In the *Register State* the clients get an ID assigned which is used in the following communication states by the clients for identification purposes.
In the *Exchange State*, clients can communicate information necessary for the simulation to the *Coordinator*. The IP used for tunnels[8] and a score used for node scheduling is sent from each client. Afterwards, the *Scenario Configs* are sent to the clients. The *Scenario Configs* include the tunnel address for each client and the decision of the node scheduler. Therefore, each client knows for which nodes he/she and others are responsible.
Finally the appropriate nodes are started by each client. In the last state (*State Distance Matrix*), the *SimulationManager* receives a step call from the user via RPC interface or the *RunLoop*. The distance matrix is then distributed via the request-reply or alternative via the publish-subscribe pattern.

[8]Controlled by a command-line parameter of the client program

5 Implementation

The following chapter provides insights into the implementation of MiniWorld. Initially, the abstract design is concretized and filled with implementation details (Section 5.1). Afterwards the format of the *Scenario Config* and implementation details are discussed. For that purpose, a *B.A.T.M.A.N. Advanced Scenario Config* for MiniWorld is presented (Section 5.2.1). The binding between the scenario config file and the internal representation is depicted in Section 5.2.2.
Following is the implementation of the node system in Section 5.3. The network configurators including connectivity checking is discussed in Section 5.4.
The implementation of the network backends is illustrated in Section 5.5. The *Command Serialization* feature is used by the *Bridged* network backends and presented together with their implementation. Afterwards, the implementation of the *Bridged* link impairment is presented in Section 5.5.2.
The implementation of the distributed mode is depicted in Section 5.6.
Creating and deploying an image as well as the modifications required by MiniWorld are introduced in Section 5.7.
The CLI is illustrated in Section 5.8. The *Event System* is used to track progress of operations and utilized the CLI to inform the user about the progress. Is is presented in Section 5.8.1. The CLI system and error checking is discussed in Section 5.8.2. Finally, the source code for an experiment in MiniWorld's distributed mode is depicted in Section 5.8.3. It demonstrates the CLI of MiniWorld.

5.1 Implemented Architecture

Figure 5.1 depicts the overall architecture of MiniWorld. The left side of the Figure points out that all functions of MiniWorld are accessible by an RPC interface. Note that there is also a Command Line Interface built on top of the RPC interface. The reason for the RPC interface is that this approach enables to easily create a web UI for MiniWorld. Furthermore, automation can be done easily.
All classes with dashed line frames, are meant to be exchangeable by the user. The upper part of the Figure shows the core of MiniWorld. The *SimulationManager* is responsible for the simulation lifecycle. This includes starting of a scenario, stopping it as well as cleaning up afterwards. The *MovementDirector*[1] is called by the *SimulationManager* for each **step**. A *step* can be triggered either automatically from the *RunLoop* or manually by the user. For that purpose, he/she has to use the CLI or RPC interface. The *MovementDirector* returns for each node the distance it has to another node. This allows the *LinkQualityModel* to decide based on the distance between two nodes whether they shall be connected. Moreover, if a connection between two peers shall be established, attributes about the link quality are returned too. It is up to the *NetworkBackend* to apply these attributes on the virtual links.
After the distance matrix and the link qualities have been obtained, the *SimulationManager*

[1]Credits for the movement patterns go to Patrick Lampe, the code is left from a university internship

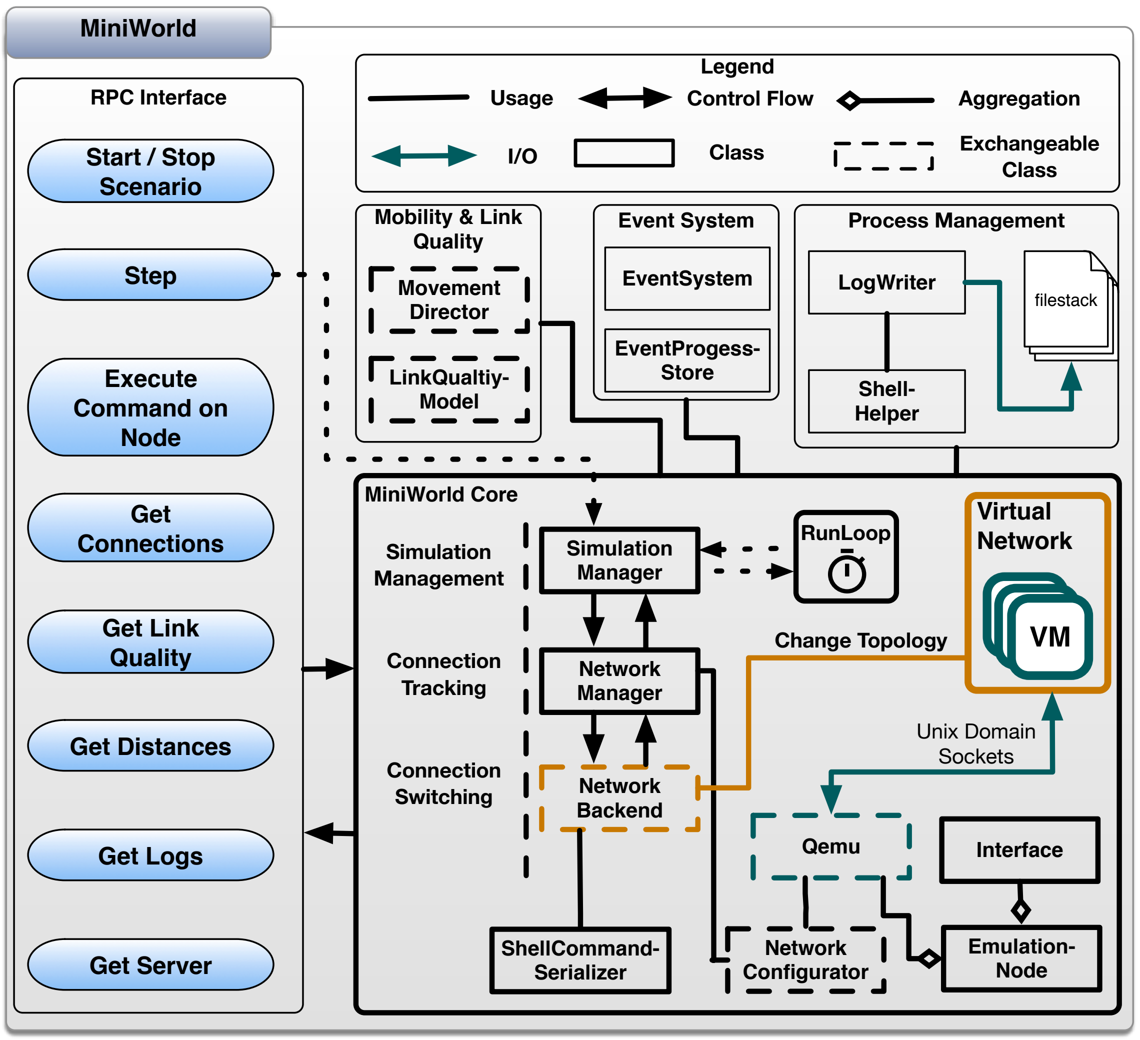


Figure 5.1: MiniWorld Architecture Implementation

calls for each connection that has to be established (according to the *LinkQualityModel*) the *NetworkBackend*. The communication between them is silently observed by the *NetworkManager* which keeps track of currently established connections. It does so, by calling specific methods of the *NetworkBackend*.
The connection tracking of the *NetworkManager* is used by the *SimulationManager* to improve performance. The *NetworkBackend* is only notified about new connections, nodes which are out of range and shall be disconnected and those there the link quality has to be adjusted. The internals of the *NetworkBackends* are discussed in Section 4.10. Note that network backends which rely on shell commands to create the virtual links, can make use of the *ShellCommandSerializer*. It allows commands to be arranged such that a sequence of commands can be executed in parallel. These commands can then either be executed in parallel, in one shell call or in batch mode for those commands who support it. See Section 4.10.2 for further information.
The connection switching has been illustrated briefly. Following is a description of the node virtualization. Each virtual node is represented by the *EmulationNode* class. A node

consists of several interfaces. The *SimulationManager* allows only interfaces of the same type to be interconnected. For other types, the *NetworkBackends* callback handlers are simply not notified.
Each node is represented by a Qemu instance. In the background KVMs are started. If the processor does not support VT, the dynamic translation capabilities of Qemu are leveraged. This is also true when a different than the native processor architecture has to be emulated. The Qemu class and the Qemu process are interconnected by a Unix Domain Socket. This gives MiniWorld control over the VM without requiring network to be set up in the VM (for example for ssh). The shell prompt of the VM is configured in the *Scenario Config*. The UDS of the Qemu process is connected to the serial console of the VM. This enables MiniWorld to view the kernel and boot log. The boot process is finished if either the boot prompt or a user configured string has been read on the serial console[2]. An additional UDS socket, connected to the Qemu Monitor, allows MiniWorld to interact with the Qemu process. The monitor connection allows for example the creation of snapshots which is leveraged to support the *Snapshot Boot Mode*.
Not relying on ssh for the node provisioning process, enables MiniWorld to configure the VM without any preconfigured network settings. Network setup commands are simply sent to the Unix Domain Socket.
Commands which shall be executed on all or only a specific node are stored inside the *Scenario Config*. An optional *NetworkConfigurator* takes care of providing each interface of a VM with an IP address. Note that there are two different kinds of network configurators available. Moreover, each *NetworkBackend* may decide to use a custom implementation, thus making the network configuration component exchangeable.
Note that even the virtualization layer can be exchanged so that containers virtualization may be added to MiniWorld.
The *NetworkConfigurators* not only provides IP addresses to the VM interfaces, they also provide a feature to test network connectivity based on the IP layer. The default connectivity check is performed with the *ping* command. Only changes in the network topology are checked (**Differential Network Checking**). This means that only new[3] links are checked for connectivity. The connectivity checking feature especially helps in the development of new network backends.
Processes, especially Qemu, is started with the help of the *Process Management* component. It takes care of starting fore- and background processes. It monitors the exit codes and redirects stdout as well as stderr to the *LogWriter*. The processes output is written to a file. The output of different processes is multiplexed by appending the output with a prefix. The *LogWriter* could be easily changed to redirect the processes stdout and stderr to a SQL database instead.
Throughout all operations in MiniWorld, the *Event System* is notified about the progress of each operation. Each node has certain events and an associated progress value. The simulation progress is then calculated from the node progresses.

[2]The VM needs to be configured such that a shell is automatically spawned on the serial console. Moreover autologin facilities have to be enabled on this console. See Section 5.7 for further instructions

[3]It would be also possible to check if tearing down a connection has been successful, but requires a ping with a timeout. This timeout needs to be so high that perfect accuracy can be achieved and slows down network checking enormous.

5.2 Scenario Config

The config system of MiniWorld is based on JavaScript Object Notation (JSON). The file format was chosen due to its simplicity and the possibility to group elements in a nested way.
There are two config files in MiniWorld. The *Config* holds static settings which are durable as long as the MiniWorld service is started. The second one is the *Scenario Config*. It bundles the describing elements of a scenario in one file.
In the following first an example *Scenario Config* for B.A.T.M.A.N. Advanced is discussed (Section 5.2.1). Finally, the binding between the textual representation and the definition of a config API via the config system is illustrated in Section 5.2.2.

5.2.1 Example Scenario Config

Figure 1 shows a *Scenario Config* for the batman advanced routing protocol which operates on the link layer. First the scenario needs a name (line 2). The name needs to be unique because it is used by MiniWorld for VM snapshots. The third line declares the number of nodes which shall be created. Line 4 introduces a new section there all entries deal with the provisioning of nodes. First the image is declared for all nodes. The config system allows customization of the entries for each node. Line 39 shows the customization of node one which has another image and more RAM allocated. In the example the first node is used as the coordinator of an experiment and needs extra scripts in the image. The same behaviour can be achieved by customizing VMs with custom images mounted inside the VM. Custom CLI parameters can be passed to Qemu in the qemu section (line 22).
In order to check whether a VM has already booted, the node provisioning system uses the regular expression declared in line 6. The commands depicted in *shell.pre_network_start* (line 7-15) are executed on the serial console of the VMs to set up B.A.T.M.A.N. advanced.
The *network* section (line 24-37) states that a chain topology shall be used for the first 30 steps. Afterwards the topology is switched to a grid of the same size. The topology files have been created with the *CORE* emulation tool. The mobility pattern is called *CORE* and enables to switch between XML defined topologies in defined time steps. Line 35 shows that the *LAN* mode shall be used. The *LAN* mode only takes into account whether nodes are connected according to the XML files. In the *WiFi* mode of the *CORE* nobility pattern, the distances between nodes are used to determine link qualities between nodes. Moreover, a topology description can also be looped.
Note that not all values in the *Scenario Config* have to be supplied. Built-in default values are taken for unspecified values. The config system has a list of expected arguments for each config option and knows whether a value is required or the default value can be taken. In the illustrated example, the connectivity checker is enabled because the default value is taken. Therefore, the IP configuration of the nodes is also enabled. Line 27 declares that only NICs with the *bat* prefix shall be provisioned with an IP address. Hence, *bat0* gets an IP address.
The *Scenario Config* has a variety of configuration options. A sample *Scenario Config* can be found in the appendix (Listing 38).

```
1  {
2    "scenario" : "batman-adv",
3    "cnt_nodes" : 128,
  "provisioning" : {
    "image": "debian_8_batman_adv.qcow2",
    "regex_shell_prompt" : "root\\@miniworld\\:\\~\\#",
    "shell" : {
      "pre_network_start": {
        "shell_cmds": [
          "ifconfig eth0 0.0.0.0",
          "modprobe batman-adv",
          "batctl if add eth0"
        ]
      }
    }
  },
  "qemu" : {
    "ram" : "256M",
    "nic" : {
      "model" : "virtio-net-pci"
    },
    "qemu_user_addition": "-hdb special_image.img"
  },
  "network" : {
    "links" : {
      "configuration" : {
        "nic_prefix" : "bat"
      }
    },
    "core" : {
      "topologies" : [
        [30, "distributed/chain_128.xml"],
        [0, "distributed/grid_128.xml"]
       ],
      "mode" : "lan"
    }
  },

  "node_details": {
    "1": {
      "provisioning" : {
        "image": "debian_8_batman_adv_experiment_coordinator.qcow2"
      },
      "qemu": {
        "ram": "1024M"
      }
    }
  }

}
```

Listing 1: B.A.T.M.A.N. Advanced Scenario Config

5.2.2 Scenario Config Binding & API

The bridge between entries in a config file and the Application Programming Interface (API) is built by decorators. Listing 5.2 shows how both fit together. A part of the *Scenario Config* is

```
{
 "network" : {
  "backend" : {
   "execution_mode" : {
    "name" : "iproute2"}}}
}
```

(a) Scenario Config

```
@customizable_attrs("network", "backend",
↪  "execution_mode", "name",
↪  default="iproute2",
↪  expected=["iproute2", "pyroute2",
↪  "brctl"])
def get_network_backend_bridged_
execution_mode(self):
    pass
```

(b) Scenario Config Implementation

Figure 5.2: Config System

illustrated on the left side. The value for the key *network→backend→execution_mode→name* is *iproute2*. The key is supplied to the decorator which is depicted on the right side of Listing 5.2. Additionally, the config system can restrict the possible values (*expected* keyword) and set a default value if the key is not supplied at all. Keys can also be marked to provide a value which is not *None*. The usage of decorators enables to form an API which is not affected by changes of keys in the config file. Instead, the name of the method which is decorated builds the API.
Adding new keys to the config system is as easy as adding a new method to it which is then accessible by all components in MiniWorld.

5.3 Node System

Figure 5.3 illustrates the node system implemented in MiniWorld. The most important object is the **EmulationNode** (middle of the Figure). It takes care of abstract node management functions such as the starting of the node virtualization layer and running the {*Pre,Post*} *Network Shell Commands*. The **EmulationNodeNetworkBackend** object decouples the network functionality from the *EmulationNode* and manages the interface for each node. The **VirtualizationLayer** class is the implementation of the node within the specific virtualization technology. Currently there is only the Qemu virtualization layer deployed. Qemu has two UDS connections: One to the serial console of a VM, the other to the *Qemu Monitor*. Both are handled by the *REPLable* mechanism.
States which need to be reset to start a new scenario can be handled by subclassing from the **StartableSimulationStateObject**. It registers itself in a garbage collector like object which cleans up the state of every registered object by calling a specific method.
The *EmulationNode*, *VirtualizationLayer* and the *EmulationNodeNetworkBackend* are part of the *NetworkBackendBootStrapper* singleton. The type of the classes are dynamically populated

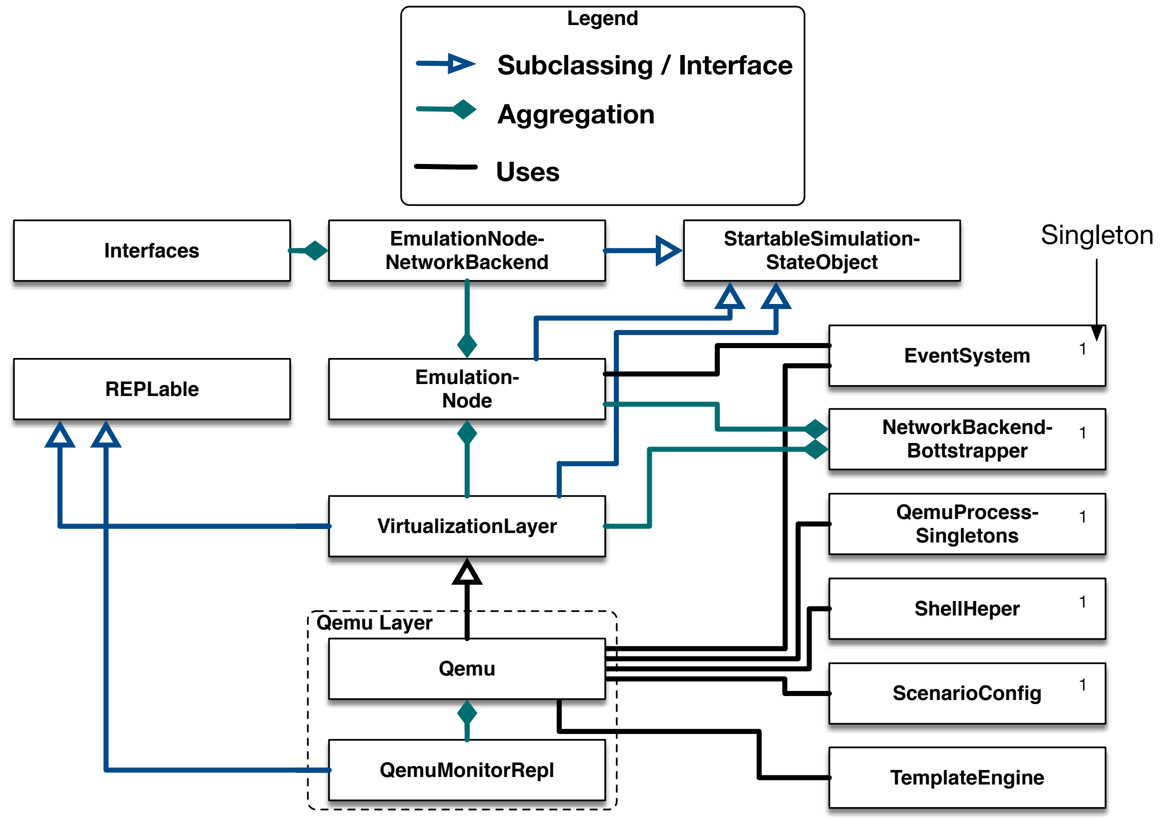


Figure 5.3: Nodes

after the *Scenario Config* has been set and used everywhere in the code to create the appropriate object. This enables to simply switch out implementations by changing the type in the *NetworkBackends* class.
Listing 2 depicts the code used for starting nodes within a specific virtualization layer.

```python
def _start(self, *args, **kwargs):
 # get local destined arguments
  flo_post_boot_script = kwargs.get("flo_post_boot_script")
  if flo_post_boot_script is not None:
      del kwargs["flo_post_boot_script"]

  self.emulation_node_network_backend.start()

  # start virtual node
  self.nlog.info("starting node ...")
  self.virtualization_layer.start(*args, **kwargs)
  self.nlog.info("node running ...")

  # notify EventSystem even if there are no commands
  es = singletons.event_system
  with es.event_no_init(es.EVENT_VM_SHELL_PRE_NETWORK_COMMANDS,
 ↪  finish_ids=[self.id]) as ev:
```

```
17        # do this immediately after the node has been started
18        self.run_pre_network_shell_commands(flo_post_boot_script)
19
20    self.do_network_config_after_pre_shell_commands()
```

Listing 2: EmulationNode Start

All arguments supplied to the *_start* method are passed on to the virtualization layer. Arguments for the *EmulationNode* itself are discarded from *kwargs*. *Flo_post_boot_script* is a file-like object containing the *Pre Network Shell Commands*.
First the *EmulationNodeNetworkBackend* is started (line 7). The *VirtualizationLayer* is started afterwards. Note that currently only the KVM/Qemu virtualization layer is implemented, but the development of a new one is straightforward due to the dynamic *NetworkBackend-BootStrapper* which supplies the type for the *virtualization_layer* class variable.
Objects specific to a node have a special logger (line 10,12) which prefixes output with the node number.
After the node has been started, the *Pre Network Shell Commands* are executed (line 18). This is handled by the *VirtualizationLayer*. In case of Qemu, the *REPLable* mechanism is used to execute the commands. The *EventSystem* is notified afterwards about the event (when the ContextManager exits). Finally the *EmulationNodeNetworkBackend* is notified that commands have been executed. It leverages this to rename the management interface inside the host. Listing 3 outlines the commands executed on node one.

```
1>>>  last_eth=$(ls -1 /sys/class/net/|grep eth|tail -n 1)
1>>>  ip link set name mgmt $last_eth
1>>>  ifconfig mgmt up
1>>>  ifconfig mgmt 172.21.0.1 netmask 255.255.0.0 up
```

Listing 3: Management Network Interface Setup By *EmulationNodeNetworkBackend*

The *1>>>* is inserted by MiniWorld to multiplex the command automation for all nodes to stdout. The management interface is always the last one. Therefore, the interface name can be fetched from */sys/class/net/* and renamed with the iproute2 command. Note that this solution works only for Linux guests with iproute2 installed. If the guest does not offer these attributes, the commands are still executed but fail. Therefore, no checking for the exit codes is done here.
The code is triggered from the *EmulationNode*. There is a callback method which allows actions to be performed for further configuration of the network. It is called after a node has been started and the *Pre Network Shell Commands* have been executed. The management setup and further customization can be achieved by exchanging the *EmulationNodeNetwork-Backend* in the *NetworkBackendBootstrapper* object[4].

[4]The object is dynamically populated with class types in the *NetworkBackends* class

5.3.1 Qemu

There are a lot of things to consider when starting a Qemu VM. MiniWorld does not rely on *libvirt*, instead it uses plain Qemu to leverage the full flexibility of the emulator. The Qemu CLI command is built from many options declared in the *Scenario Config*. The number of NICs has to be incorporated into the Qemu command. For that purpose, MAC addresses need to be created which are derived from the node id. A MAC address has 48 bits. The first byte is used for the node class, which is an integer representing an interface type. The second bit is used for the number of the interface. Therefore, $2^8 = 256$ interfaces of the same type are currently supported in the MAC address scheme. The remaining 32 bits are used to encode the node id ($2^{4*8} = 4,294,967,296$).
For the default image, an overlay image is generated to make use of the Copy On Write mechanism. For this the *QCOW2*[5] image format is used which has the COW acronym in its name. For each node image and for images defined to have a COW layer[6], an *QCOW2* image is created with the *qemu-img* command. The base file used for the COW layer is called *backing file*.

```
qemu-img create -b <image> -f qcow2 <overlay>
```

Listing 4: Qemu Overlay Image Command

The mechanism enables to define one image used by all nodes without affecting each other by providing a common read only layer to all nodes. Due to the write layer, changing the *backing file* requires to define either the new *write layer* in the *Scenario Config* or manually modifying the *backing file* as illustrated in Section 5.7. The command to create the overlay image is depicted in Listing 4.
Booting a VM is easy, but checking if the start process is finished, is not. There are three *Boot Modes* implemented. A custom string can be printed to the serial console's output by writing for example to */proc/kmsg*. */etc/rc.local* is a good place for Linux systems to signal MiniWorld that the boot process is finished. This string is called *Boot Prompt*[7] in the following. Some OpenWRT versions for example print `"procd: - init complete -"` to the kernel log.
The second boot mode follows the idea of mimicking a human being. For that purpose, pressing the enter button is simulated by sending `\n` to the serial console of the VM in predefined time intervals. If the *Shell Prompt*[8] is detected in the output of the UDS, the VM has booted. Both modes rely on a python 2 backport of the *selectors*[9] package to provide efficient I/O multiplexing. The default select system call is limited to 1024 file descriptors and is not as fast as for example the *epoll* mechanism. For BSD, *kqueue* is the most efficient selector. The package provides the best selector in terms of speed independent of the OS. The described boot modes are called **Selectors Boot Prompt** and **Selectors Shell Prompt**. There is another implementation using the *Boot Prompt* based on the *pexpect*[10] package. It is a implementation mimicking the original expect command which can be used to automate

[5]`https://en.wikibooks.org/wiki/QEMU/Images#Copy_on_write`. Last viewed on 27.11.2016.
[6]Key provisioning→overlay_images in the *Scenario Config*.
[7]This string is defined in provisioning→regex_boot_completed in the *Scenario Config*.
[8]The string returned by the Shell if enter is pressed.
[9]https://docs.python.org/3/library/selectors.html. Last viewed on 27.11.2016.
[10]`https://pexpect.readthedocs.io/en/stable/`

interactive programs such as telnet. The **Pexpect Boot Prompt** mode was the first in MiniWorld, but is superseded by the selectors implementation due to the limitations[11] of the *select* system call.
The boot time times of full system virtualized VMs are very high in contrast to lightweight virtualization. A special boot mode called **Snapshot Boot Mode** tries to reduce these times. Snapshots are taken from the VMs. For that purpose, {*savevm,loadvm*} <*snapshot_name*> commands are sent to the Qemu Monitor. If a snapshot cannot be loaded, the string *Device '.*' does not have the requested snapshot* is used to detect an error. In that case, a VM is booted normally. The snapshot name is taken from the name declared in the *Scenario Config*. The snapshot is linked to the Qemu process, therefore it is stored in the *QemuProcessSingletons* class for each node. Classes execute commands with the help of the *ShellHelper*. The processes are then managed by the *ShellHelper* singleton. This includes the multiplexing of stdout and stderr to a log file as well as the shutting down the process if the scenario is stopped. In the *Snapshot Boot Mode*, the Qemu process has to be kept alive to keep the snapshot. The process if first shut down, if a new VM with a different scenario name is started.

5.4 NetworkConfigurators

NetworkConfigurators provision VMs by means of IP and enable network connectivity checking. The setting of IP addresses inside a Linux VM could be implemented easily in the *EmulationNodeNetworkBackend*. This is done for the *management* interface. The configurator for the *Bridged LAN* network backend requires the knowledge of the current connections. It then iterates over the connections and puts the interfaces of the connection into a separate subnet. Moreover, the IP addresses of connected nodes are remember to provide the connectivity checking mechanism. This can be achieved easier from a central place instead of implementing this inside a *EmulationNodeNetworkBackend*. The connectivity checker function is a bash function which can be supplied via the *Scenario Config*, but is limited to the usage of IP, since IP addresses are used to identify the peers. The described network configurator is implemented by the *NetworkConfiguratorP2P*. It takes into account the nature of *Bridged LAN* network backend where a NIC is used to represent a connection between nodes.
In contrast to the aforementioned configurator, the *NetworkConfiguratorSameSubnet* provides each NIC with an IP address from the same subnet[12]. Note that a different subnet is provided for each interface. The same applies if there are two interfaces of the same kind. The *Bridged LAN* network backend would not work with the *NetworkConfiguratorSameSubnet*, since all NICs would be on the same subnet and hence the kernel would route packets only through one interface.
The connectivity checking is performed in parallel by default and fails if a predefined timeout is not enough. The standard connectivity checker is the *ping* command. Links are assumed to be bidirectional. Therefore, connections are only *pinged* from one side of a connection. This holds also if two nodes are distributed among different servers. It reduces the required check commands by half.

[11] `https://github.com/pexpect/pexpect/issues/47`. Last viewed on 27.11.2016
[12] CIDR notation can be supplied via the *Scenario Config*.

5.5 NetworkBackends

The implementation of the network backends is mainly discussed by showing the commands required to start the nodes and set up the network. The commands are based on a *Chain 2* topology if not stated differently.
First the *VDE* network backend is introduced in Section 5.5.1 to the reader. Following is the *Bridged* network backend in Section 5.5.2 which consists of the *Bridged LAN* and the *Bridged WiFi* network backend. The implementation of the *WiFi* network backend is discussed in Section 5.5.3.

5.5.1 VDE

The *VDE* network backend requires modifications to the Qemu class. *QemuVDE* handles the creation of the NIC arguments for the *VDESwitch*. The *VDE* and Qemu commands are shown in Listing 5.

```
1 vde_switch -sock "/tmp/MiniWorld/vde_switch_1_2_1" -M
  ↪ /tmp/MiniWorld/vde_switch_mgmt_1_2_1 -hub
2 vde_switch -sock "/tmp/MiniWorld/vde_switch_1_10_1" -M
  ↪ /tmp/MiniWorld/vde_switch_mgmt_1_10_1 -hub
3 qemu-system-x86_64 -enable-kvm -cpu host -m 1024M -serial
  ↪ unix:/tmp/MiniWorld/qemu_1.sock,server -monitor
  ↪ unix:/tmp/MiniWorld/qemu_monitor_1.sock,server -nographic
4 -device virtio-net-pci,netdev=net0,mac=02:01:00:00:00:01 -netdev
  ↪ vde,id=net0,port=2,sock=/tmp/MiniWorld/vde_switch_1_2_1
5 -device virtio-net-pci,netdev=net1,mac=0a:01:00:00:00:01 -netdev
  ↪ vde,id=net1,port=2,sock=/tmp/MiniWorld/vde_switch_1_10_1
6 -hda "openwrt-x86-miniworld_dev_v55_overlay_1.img"
```

Listing 5: Qemu And VDE Commands (Node 1 only)

For each interface a *VDESwitch* is started (lines 1-2). Qemu network backends consist of two parts: One is responsible for providing the virtual NIC inside the guest, the other part is the backend which interacts with the NIC[13]. The *-dev* command line switch defines the virtual NIC (lines 4-6). *Virtio-net-pci* is a paravirtual driver which improves network performance. The *-netdev* argument (lines 4-6) defines that the *VDESwitch* whose UDS is located at */tmp/MiniWorld/vde_switch_1_2_1*[14] shall be used to connect the VM to the network. There are two NICs in the command: The first is the actual interface, the second the management interface. Both switches are connected to port 2.
Listing 6 shows the commands sent to the UDS of the appropriate *VDESwitch* to connect node 1 and 2.

```
1 # set total number of ports
2 1>>> port/setnumports 65537
```

[13] http://wiki.qemu.org/Documentation/Networking. Last viewed on 27.11.2016.
[14] The prefix of the switches uses the following format:<node_id>_<interface_type>_<interface_nr>.

```
3  2>>> port/setnumports 65537
4
5  # enable colors
6  1>>> port/setcolourful 1
7  2>>> port/setcolourful 1
8  # set color for port 2
9  1>>> port/setcolour 2 2
10 2>>> port/setcolour 2 2
11
12 # get used ports
13 1>>> port/print
14 2>>> port/print
15
16 # connect nodes
17 (1, 2)>>> dpipe vde_plug -p 32 /tmp/MiniWorld/vde_switch_1_2_1 = wirefilter
    ↪ -M /tmp/MiniWorld/wirefilter_1_2_1_2_2_1.sock -l 100 = vde_plug -p 14
    ↪ /tmp/MiniWorld/vde_switch_2_2_1
18
19 # vlan
20 1>>> vlan/create 2
21 2>>> vlan/create 2
22 1>>> port/setvlan 32 2
23 2>>> port/setvlan 14 2
24
25 # adjust link quality
26 1_2_1,2_2_1>>> loss 0
```

Listing 6: VDE Network Setup

First the number of ports are set (lines 1-3). Second, the color mode is activated and the links are colored with the number of interface type[15] (lines 5-10). A *Wirefilter* is used to connect both nodes (line 17). The ports 32 and 14 are randomly chosen from the free ports. Used ports are read from the output of the *port/print* command. The connection is created with 100% loss. The *loss 0* command finally enables both nodes to communicate (line 26). The output of some *VDESwitch* commands is illustrated in Listing 7. Each successful command ends with *1000 Success* and is used by the *REPLable* mechanism.

```
3>>> $ port/setvlan 5711 10
3>>> 1000 Success

vde$
3>>> $ port/showinfo
3>>> 0000 DATA END WITH '.'
Numports=65537
HUB=true
```

[15] The color for the mesh interface is 2.

```
COLOURFUL=true
counters=false
.
1000 Success

vde$

3>>> $port/print
3>>> 0000 DATA END WITH '.'
Port 0002 untagged_vlan=0000 ACTIVE - Unnamed Allocatable
 Current User: root Access Control: (User: NONE - Group: NONE)
 colour:            10
  -- endpoint ID 0003 module unix prog   : QEMU user=root PID=52499
↪   SSH=10.79.74.146
.
1000 Success

vde$
```

Listing 7: VDE REPL Commands Output

The *Shell Prompt* is *vde$* and used to wait until the output of a command is ready. The output of *port/showinfo* gives the number of ports. The implementation of the virtual nodes that power the *Hub* and *Management* are not further discussed.

5.5.2 Bridged

The bridged network backend offers two implementations: The first is the *Bridged LAN* network backend, the second the *Bridged WiFi* network backend. Both implementations share as many code as possible. The integration of both network backends within the *NetworkBackendBootStrapper* is presented in Figure 5.4. On the top right side of the Figure the connection, switch and tunnel types are shown. The bottom right side illustrates the network backend and the IP provisioner. The top left corner points out the integration of the used virtualization technology. Finally, the virtual nodes for the *Hub* and *Management* interface are shown in the bottom left corner. The beige components use *dynamic subclassing* which is explained later in this Chapter.
The components in the *NetworkBackendBootstrapper* are dynamically populated by the needs of a network backend in the *NetworkBackends* class. This holds for all network backends. The types are then used everywhere in the code to create the actual instances. Figure 5.5 outlines the integration of both network backends in a single derivation hierarchy. The base class is *NetworkBackendDummy*. It defines methods which need to be implemented by subclasses to provide a custom *NetworkConfigurator* and the *InterfaceFilter*. The *NetworkBackendNotifications* class defines the interface for the communication with the *SimulationManager*.
Remember that the *Bridged LAN* network backends requires the prior knowledge of the number of connections and is therefore only usable in conjunction with the *Core Mobility Pattern*. This is abstracted by the *NetworkBackendStatic* class which examines all CORE topology files and calculates the maximum number of connections for each node. The

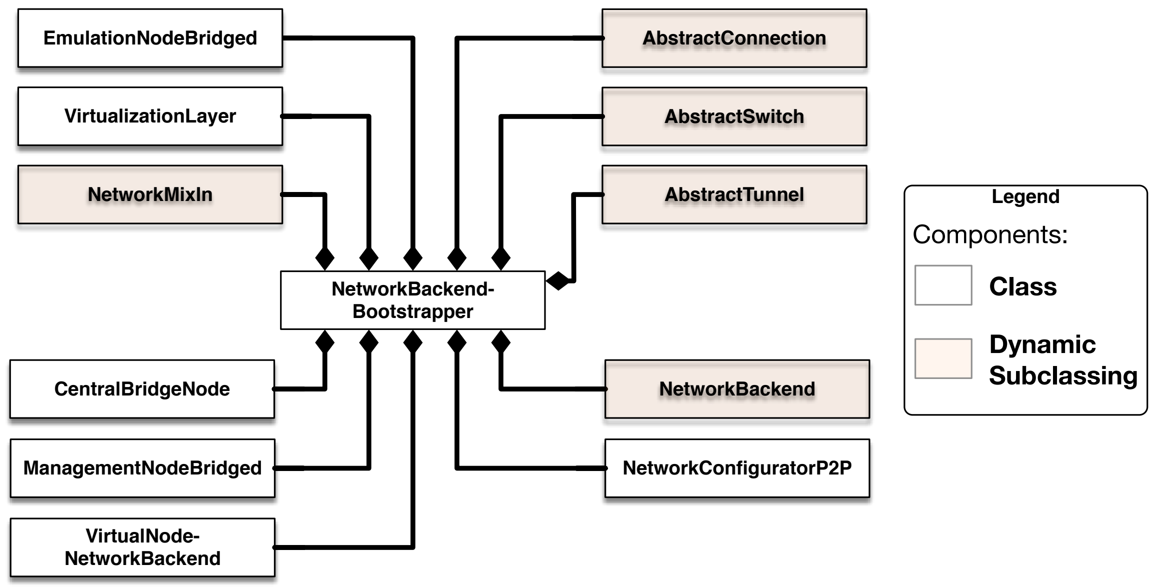


Figure 5.4: NetworkBackendBootStrapper Bridged

NetworkBackendDynamic is an empty class at the time of writing.
There are three modes implemented for both networks which are introduces in Section 5.5.2. These concrete implementations are depicted in beige color. Each mode requires either the *NetworkBackendBridgeMultiDevice* or *NetworkBackendBridgeSingleDevice* as base class. Instead of creating $2 * 3$ classes, **dynamic subclassing** is used. For that purpose, each class is defined and created from within a method and therefore does not live in the global namespace. The base class is dynamically determined depending on the settings of the *Scenario Config*. *Dynamic subclassing* is depicted with dashed lines in Figure 5.5.
NetworkBackend is the first class which derives dynamically from either *NetworkBackendStatic* or *NetworkBackendDynamic*. *NetworkBackendBridgedDummy* then uses the *NetworkBackend* method to get the appropriate base class dynamically. The actual implementations of the network backends are *NetworkBackendBridgedMultiDevice* (*Bridged Lan*) and *Network-BackendBridgedSingleDevice* (*Bridged WiFi*). The names *single* and *multi* express that either connections are multiplexed by a single NIC or one Network Interface Card per connection is required.
Not only the network backends use *dynamic subclassing*. The class diagram of switches and connections is presented in Figure 5.6. The bridge is the same for both network backends. Only the connection management differs

In the following, first the *execution modes* are illustrated. Then implementation details of both network backends are discussed.

Execution Modes

There are different possibilities by which the virtual network can be created and controlled. **Brctl** is a Linux command which handles bridge operations. **Pyroute2** is a python library which uses netlink sockets to interact directly with the kernel. There are two different ways of using the library. One is to use the *IPRoute* object which fetches information from the kernel for every call. There is also a batch mode such that commands can be grouped into

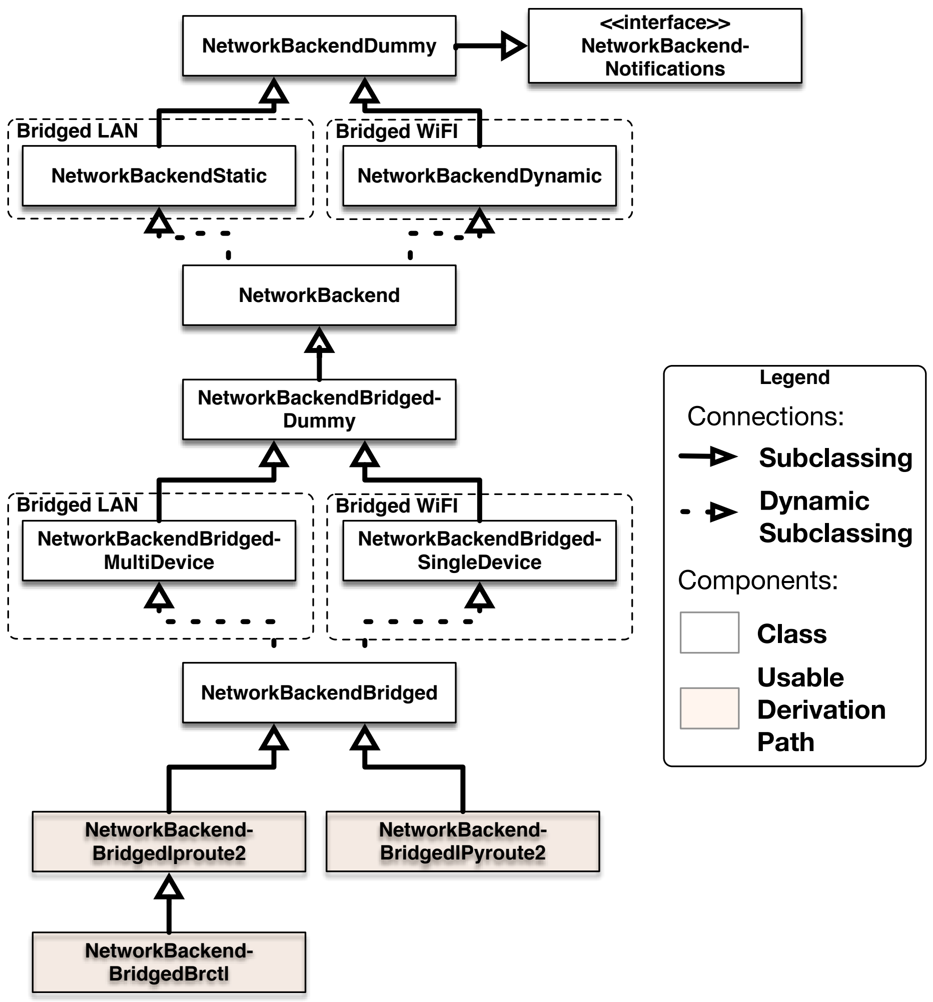


Figure 5.5: Bridged Network Backend Class Diagram

one call. Another approach is to use the transactional IPDB object which keeps in sync with the kernel by listening to netlink broadcast messages. **Iproute** is the successor of tools such as ifconfig and brctl.
Brctl in contrast to *Iproute2* worked out of the box on all tested machines. Even though it does not offer a batch mode like *Iproute2* does, it is included for situations where ease of use is more important than performance.
Table 5.1 outlines for each operation the equivalent commands for each virtual network setup. Rows 1 and 2 are commands for the Linux shell whereas the 3. and 4. rows depict python code. Variables are shown as <variable>. The python method get_iface_idx() is equal to a call to pyroute2.IPRoute().link_lookup(ifname=<interface>)[0]. Since this requires communication with the kernel for every call, an interface cache is fetched once from the kernel after all bridges have been created. The cache is required to get the index of an interface. Note that calls are not made directly to pyroute2.IPRoute, instead the pyroute2.IPBatch object is used to provide a batch mode which is turned on by default.
The commands are used for creating a bridge, adding an interface to a bridge, changing the state of an interface and for transforming a bridge into a hub (essential for the *Hub* interface). Creating a tunnel and deleting interfaces is solely done with *iproute2*.

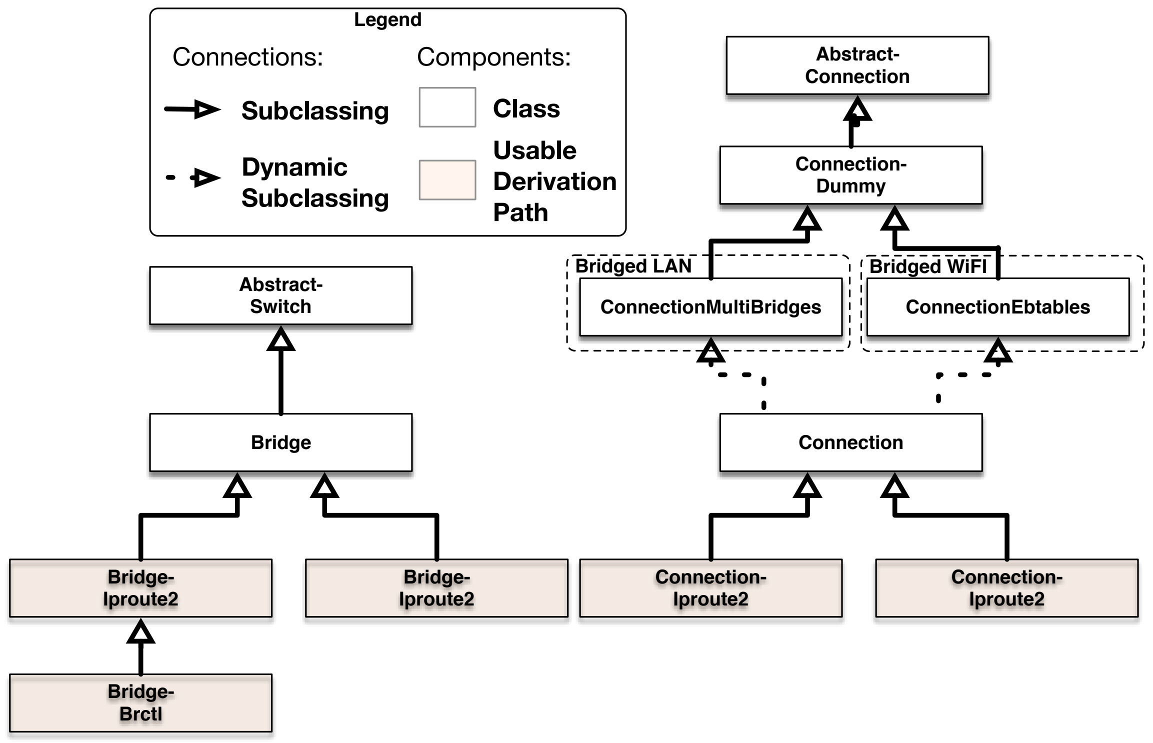


Figure 5.6: Bridged Network Backend Switch And Connection Class Diagram

The connection switching commands also differ in way the are executed. *Brctl* commands are executed sequentially. Communication with the kernel produces overhead for each command. *Iproute2* is able to execute all commands at once (*batch* mode). *Pyroute2* commands are executed within a commit, thus allowing rollbacks if an error occurs. The performance of these commands for connection switching is evaluated in Section 6.4.4.

Execution Options

Listing 8 depicts the commands used to setup a *Chain 2* topology including link quality impairment. The semantics of the *tc* and *ebtables* commands can be ignored, since they are introduced later. The only thing that is important is that the commands in one section are independent of each other. Therefore, they can be executed in an arbitrary order.
The commands are taken from the */tmp/MiniWorld/logs/network_backend_shell_commands.txt* file. The Listing points out how commands are grouped into parallelizable events.

```
#Parallelizable commands for: ebtables, ebtables_create_chain:
ebtables --concurrent  -N wifi1 -P DROP

#Parallelizable commands for: ebtables, ebtables_redirect:
ebtables --concurrent  -A FORWARD --logical-in wifi1 -j wifi1

#Parallelizable commands for: ebtables, ebtables_commands:
```

Semantics	Command (iproute2 / brctl / pyroute2 IPRoute / pyroute2 IPDB)
Create Bridge Device	ip link add name <bridge> type bridge
	brctl addbr <bridge>
	pyroute2.IPRoute().link("add", kind="bridge", ifname=<bridge>)
	pyroute2.IPDB().create(kind='bridge', ifname=<bridge>)
Set Hub Mode	ip link set dev <bridge> type bridge ageing_time 0
	brctl setageing <bridge> 0
	pyroute2.IPRoute().link("set", index=get_iface_idx(<bridge>), ageing_time=0)
	<bridge_obj>.set_br_ageing_time(0)
Enslave NIC	ip link set dev <interface> master <bridge>
	brctl addif <bridge><interface>
	pyroute2.IPRoute().link('set', index=get_iface_idx(<interface>),
	master=get_iface_idx(<bridge>))
	<bridge_obj>.add_port(<bridge_obj>['index'])
Change NIC State	ip link set dev <interface> (up,down)
	-
	pyroute2.IPRoute().link('set', index=get_iface_idx(<interface>), state=<state>)
	<tap_obj>.up(), <tap_obj>.down()
GreTap Tunnel	ip link add <tunnel_name>type gretap remote <remote_ip>key <key> nopmtudisc
Delete Interface	ip link del <interface>

Table 5.1: Connection Switching Commands

```
ebtables --concurrent  -I wifi1 -i tap_00001_1 -o tap_00002_1 -j mark --set-mark 1 --mark-target ACCEPT
ebtables --concurrent  -I wifi1 -i tap_00002_1 -o tap_00001_1 -j mark --set-mark 1 --mark-target ACCEPT

#Parallelizable commands for: connection, link_shape_add_child:
tc qdisc replace dev tap_00001_1 root handle 1:0 htb
tc qdisc replace dev tap_00002_1 root handle 1:0 htb

#Parallelizable commands for: connection, link_shape_add_class:
tc class replace dev tap_00001_1 parent 1:0 classid 1:1 htb rate 11485.8795215kbit
tc qdisc replace dev tap_00001_1 parent 1:1 handle 10: netem delay 7.40ms 0.74ms 25%
tc class replace dev tap_00002_1 parent 1:0 classid 1:1 htb rate 11485.8795215kbit
tc qdisc replace dev tap_00002_1 parent 1:1 handle 10: netem delay 7.40ms 0.74ms 25%

#Parallelizable commands for: connection, link_shape_add_filter:
tc filter replace dev tap_00001_1 parent 1:0 protocol all handle 1 fw flowid 1:1
tc filter replace dev tap_00002_1 parent 1:0 protocol all handle 1 fw
↪ flowid 1:1

#Parallelizable commands for: bridge, bridge_add:
ip link add name wifi1 type bridge

#Parallelizable commands for: bridge, bridge_parallel:
ip link set dev wifi1 group 2
ip link set dev wifi1 type bridge ageing_time 0
ip link set dev tap_00001_1 master wifi1
ip link set dev tap_00001_1 up
ip link set dev wifi1 up
ip link set dev tap_00002_1 master wifi1
ip link set dev tap_00002_1 up
```

Listing 8: ShellCommandSerializer Chain 2

There are several options how the commands can be executed. The first is sequential. This includes no **Execution Option** at all.
The **Parallel** option executes all commands from an event in parallel. Instead of letting each worker execute one command, they can be grouped to one. For that purpose they are chained together with *sh -c "cmd1; cmd2"* such that the whole command fails if one of it subcommands fails too. The option has been introduced since the python *subprocess* module slows down operations while performing many shell calls. The option is called **Single Shell Call**. Some commands support a **Batch** mode. Examples are *ip* and *tc* from the *iproute2* package. The batch version of the commands are illustrated in Listing 9 and 10 respectively.

```
ip -d -batch - <<<"cmd1\ncmd2\n...cmd3"
```

Listing 9: Ip Batch Mode

```
tc -d -batch - <<<"cmd1\ncmd2\n...cmd3"
```

Listing 10: Tc Batch Mode

The *Execution Options* can also be combined. The *Batch* mode is prioritized over the *One Shell Call* mode which is prioritized over *Parallel* execution since it removes parallel execution.

Command Serialization

The *Bridged* network backends are the only ones which require shell commands for their network setup and link impairment. Hence it is introduced in the following before the implementation of both *Bridged* network backends is introduced to the reader.
The integration of the *NetworkBackend* by means of callback methods provided by the *SimulationManager* requires *NetworkBackends* to react to each event. Shell commands to create switches, connections and tunnels could be directly executed from the appropriate callback method. Another possibility is to delay the command execution and to remember only the commands that need to be executed. Although, this requires prior knowledge of dependencies to order the commands such that they can be executed in the correct order. Such a dependency is for example that a bridge needs to exist before any operations on it can be performed.
The **Command Serialization** feature enables commands to be executed in an order which is independent of the insertion order. Commands are serialized such that commands can be executed in parallel. Moreover, the number of commands is reduced by filtering out duplicates. Incorporating dependencies allows commands to be executed in parallel. For that, commands are required to be grouped such that dependencies are resolved. There are a few terms that need to be introduced to the reader. First, there is the concept of an **Event**. It is used to group similar commands. Multiple events make up a **Group**. The order of the events inside the group is called **Event Order**.

```
>>> scs=ShellCommandSerializer()

>>> scs.set_event_order("bridge", ["bridge_add", "bridge_add_if"])
#                                                ^
#                                           event_order
>>> scs.set_event_order("connection", ["state_change"])
>>> scs.set_group_order(["bridge", "connection"])
>>> # naming:
>>> #                        ^             ^
>>> #                      group         event
>>> scs.add_command("bridge", "bridge_add_if", _, "brctl addif br_foobar
 ↪  eth0", _)
>>> scs.add_command("bridge", "bridge_add_if", _, "brctl addif br_foobar
 ↪  eth1", _)
>>> scs.add_command("bridge", "bridge_add",    _, "brctl addbr br_foobar"
 ↪  , _)

>>> scs.add_command("connection", "state_change", _, "ifconfig eth0 up", _)

>>> print scs.get_all_commands()
['brctl addbr br_foobar', 'brctl addif br_foobar eth0', 'brctl addif
 ↪  br_foobar eth1', 'ifconfig eth0 up']
# run commands with one threaded per virtual core
>>> scs.run_commands(self, max_workers=None)
```

Listing 11: ShellCommandSerializer Example

Listing 11 shows the usage of the object. There are two groups: bridge and connection (lines 3,6). There are two events for the bridge and one for the connection group. Note that the >>> notation used inside the Listing depicts the python *doctest* format [16] and is not the prefix of a node id. The *Group Order* is set in line 7. Afterwards commands are added to the events in their respective group (lines 11-13,15). Note that arguments used for logging are depicted with a _ symbol for clarity. The output in line 17 points out the dependencies between the commands: First the bridge is created. Then the commands from the event *bridge_add_if* and finally the command from the *state_change* event of the *connection* group is printed. The *ShellCommandSerializer* object is used especially by the network backends to improve performance. Commands from one event are considered to be independent of each other and can be executed in parallel by multiple threads or sequentially (line 20). Even in the sequential mode, it has the advantage of removing duplicate commands. Some commands may support a batch mode which can be fed by the *ShellCommandSerializer* too.

Distributed Mode

Nodes are interconnected on the link layer with GRE by default. Both sides of a connection have to establish a tunnel. The tunnel is added to a bridge on both nodes. According to [6], problems arise for packets with a size greater than the Maximum Transmission Unit (MTU). The MTU is assumed to be equal for all nodes on the same broadcast domain. By default, Path MTU Discovery (PMTUD) is enabled for both gretap tunnels as well as for Linux (systemwide setting). The consequence is that the Don't Fragment (DF) bit is set in IP packets. Peers on a path which have a lower MTU, report with Internet Control Message Protocol (ICMP) to their previous hop their supported MTU size so that the MTU can be adjusted along the path. But since bridging operates on the link layer and PMTUD is a network layer mechanism, the bridge silently drops frames with a too large MTU. If a gretap device is added to a bridge, the bridges MTU is lowered to that of the gretap device (1459) [6], but the NICs inside the VMs still has a default MTU of 1500. The problem is also known to CORE[17].
MiniWorld solves the MTU problem by setting the correct MTU inside the VMs.
GRE tunnels are distinguished by IDs. Since the coordination of common IDs between two connected nodes introduces additional overhead, a *Pairing Function* is used to uniquely map 2 node IDs to one integer. The resulting ID is used as tunnel ID. The *Szudzik* [18] pairing function requires for the pairing of two 16 bit integers 32 bit for the unique ID.
Virtual LAN (VLAN) and Virtual Extensible LAN (VXLAN) are also implemented but showed problems with the Docker containers. Moreover, VLAN provides only 12 bit for the tunnel ID which is only sufficient for a small scenario. VXLAN in contrast uses 24 bit for the tunnel IDs.

Bridged LAN

Listing 12 shows the start command for Qemu issued by the *Bridged LAN* network backend.

[16]`https://docs.python.org/2/library/doctest.html`
[17]`https://github.com/gregtampa/coreemu/issues/93`. Retrieved on 07.12.2016.
[18]`www.szudzik.com/ElegantPairing.pdf`. Last viewed on 11.12.2016

```
1 2>>> qemu-system-x86_64 -enable-kvm -cpu host -m 1024M -serial
   ↪ unix:/tmp/MiniWorld/qemu_2.sock,server -monitor
   ↪ unix:/tmp/MiniWorld/qemu_monitor_2.sock,server -nographic
2 -device virtio-net-pci,netdev=net0,mac=02:01:00:00:00:02 -netdev
   ↪ tap,id=net0,ifname=tap_00002_1,script=no,downscript=no
3 -device virtio-net-pci,netdev=net1,mac=02:02:00:00:00:02 -netdev
   ↪ tap,id=net1,ifname=tap_00002_2,script=no,downscript=no
4 -device virtio-net-pci,netdev=net2,mac=0a:01:00:00:00:02 -netdev
   ↪ tap,id=net2,ifname=tap_00002_3,script=no,downscript=no
5 -hda "/tmp/MiniWorld/openwrt-x86-miniworld_dev_v55_overlay_2.img"
```

Listing 12: QemuTap Start (Node 2)

The Qemu network backend is *tap* (lines 2-4) and is set with the *-netdev* command line switch. The tap device names are also declared in the command. A subclass of *EmulatinoNodeNetworkBackend* creates the MiniWorld interfaces depending on the maximum number of connections a node has during the scenario. The creation of the command line argument for the network interfaces of Qemu is handled by the *QemuTap* class (subclass of *VirtualizationLayer* in Figure 5.4).
The network backend requires the *NetworkConfiguratorP2P* to configure point-to-point links and overwrites the configurator defined in the *Scenario Config*. Moreover, the *InterfaceFilter AllInterfaces* allows each interface to be connect with any other. The final decision which interface is used for a connection is up to the *Bridged LAN* network backend.
In a *Chain 3 Topology*, node 2 has a connection to both neighbours. Therefore it has 2 NICs plus one for the management network. The *iproute2* commands in batch mode to create the network topology are depicted in Listing 13. Note that the command is the *iproute2* batch mode which reduces connection switching times. The commands used to set up the network topology are written to */tmp/MiniWorld/logs/network_backend_shell_commands.txt* and sorted by *group* and *event* through the *ShellCommandSerializer*.

```
ip -d -batch - <<<"link set dev tap_00001_1 up
link set dev tap_00002_1 up
link set dev tap_00002_2 up
link set dev tap_00003_1 up
link add name br_00001_00002 type bridge
link add name br_00002_00003 type bridge
link set dev br_00001_00002 group 2
link set dev br_00001_00002 type bridge ageing_time 0
link set dev tap_00001_1 master br_00001_00002
link set dev br_00001_00002 up
link set dev tap_00002_1 master br_00001_00002
link set dev br_00002_00003 group 2
link set dev br_00002_00003 type bridge ageing_time 0
link set dev tap_00002_2 master br_00002_00003
link set dev br_00002_00003 up
link set dev tap_00003_1 master br_00002_00003"
```

Listing 13: Bridged LAN Network Backend Chain 3 Setup

Note that the link quality impairment is introduced in Section 5.5.2 first.

Bridged WiFi

The *Bridged WiFi* network backend allows any number of connections to be multiplexed over a single NIC. In contrast to the *Bridged LAN* network backend only one NIC per MiniWorld interface is used to build the Qemu command.
The *InterfaceFilter* is *EqualInterfaceNumbers* which allows only equal interfaces to be connected. This prevents the *Bridged WiFi* backend from being called for other device combinations of two nodes by the *SimulationManager*. The used configurator is *NetworkConfigurator-SameSubnet*.
The commands to set up the *Chain 3* topology are illustrated in Listing 14.

```
ebtables="ebtables --concurrent --atomic-file
 ↪ /tmp/MiniWorld/ebtables_atommic"

# clear tables
$ebtables --atomic-init
$ebtables --atomic-commit

# create chains
$ebtables --atomic-save
$ebtables -N wifi1 -P DROP
$ebtables -A FORWARD --logical-in wifi1 -j wifi1
$ebtables --atomic-commit

# layer2 firewall
$ebtables -atomic-save
$ebtables -I wifi1 -i tap_00001_1 -o tap_00002_1 -j mark --set-mark 1
 ↪ --mark-target ACCEPT
$ebtables -I wifi1 -i tap_00002_1 -o tap_00001_1 -j mark --set-mark 1
 ↪ --mark-target ACCEPT
$ebtables -I wifi1 -i tap_00002_1 -o tap_00003_1 -j mark --set-mark 2
 ↪ --mark-target ACCEPT
$ebtables -I wifi1 -i tap_00003_1 -o tap_00002_1 -j mark --set-mark 2
 ↪ --mark-target ACCEPT
$ebtables --atomic-commit

# change network topology
ip -d -batch - <<<"link add name wifi1 type bridge
link set dev wifi1 group 2
link set dev wifi1 type bridge ageing_time 0
link set dev tap_00001_1 master wifi1
```

```
26 link set dev tap_00001_1 up
27 link set dev wifi1 up
28 link set dev tap_00002_1 master wifi1
29 link set dev tap_00002_1 up
30 link set dev tap_00003_1 master wifi1
31 link set dev tap_00003_1 up"
```

Listing 14: Bridged WiFi Network Backend Chain 3 Setup

For illustration purposes, portions of the *ebtables* command are abstracted by the *ebtables* shell variable.
Ebtables is used in the concurrent mode so that multiple commands can be executed in parallel. Moreover, the batch mode is leveraged. The CLI switch *–atomic-save* copies the current *ebtable* rules from the kernel into the file provided by *–atomic-file*. This enables MiniWorld to switch only differences between network topologies. The *–atomic-commit* copies the rules from the *atomic file* to the kernel.
For each interface in MiniWorld one bridge is created (line 22). Furthermore, the *NetworkBackendBridgedSingleDevice* creates for each bridge a new chain (line 9). The default policy is to discard bridging of frames. Moreover, frames received by a bridge are redirected to the appropriate chain. The *ConnectionEbtables* is responsible of controlling which interface is allowed to communicate with another. For that purpose it inserts *ebtable* rules based on the name of the tap interfaces. For example *tap_00001_1* is allowed to communicate with *tap_00002_2* and vice versa (lines 15-16). The *–mark-target ACCEPT* means that frames shall be bridged. Moreover, the connections are **marked** internally in the Linux kernel so that for each connection different link qualities can be applied.
The functioning of *ebtables* and *tc* has already been illustrated in Section 2.4. In addition the Figure from the aforementioned Section, Figure 5.7 highlights the places where the *ebtables* firewall and connection marking is used. Based on this connection marking in the *FORWARD* chain of the *filter* table, the link quality is applied by a *tc* QDisc (bottom right corner).

Link Quality Models

In the following, the link quality impairment is illustrated for both network backends. First the more complex **Bridged WiFi** network backend impairment is shown.
Figure 5.8 provides a graphical overview of the *tc* commands and the hierarchy which is setup by them. The Figure outlines the *tc* commands for node 2 on the right side. The *Handles* are illustrated in the middle of the Figure. The *Root QDisc* is HTB. It is a *classfull QDisc* used to shape the bandwidth. It is a very easy to use QDisc, since only the bandwidth has to be supplied. Most other QDiscs require more parameters and are more complex to setup. Node 2 has two connections: One to node 1 and one to node 3. Therefore, 2 classes are required to offer a different link impairment for the connections. The commands on the ride side of Figure 5.8 belong to the connection between node 2 and node 3. For both nodes a class is created with a different bandwidth. More advanced link emulation is offered by *netem* which allows to simulate delay, packet reordering, loss and much more. Currently,

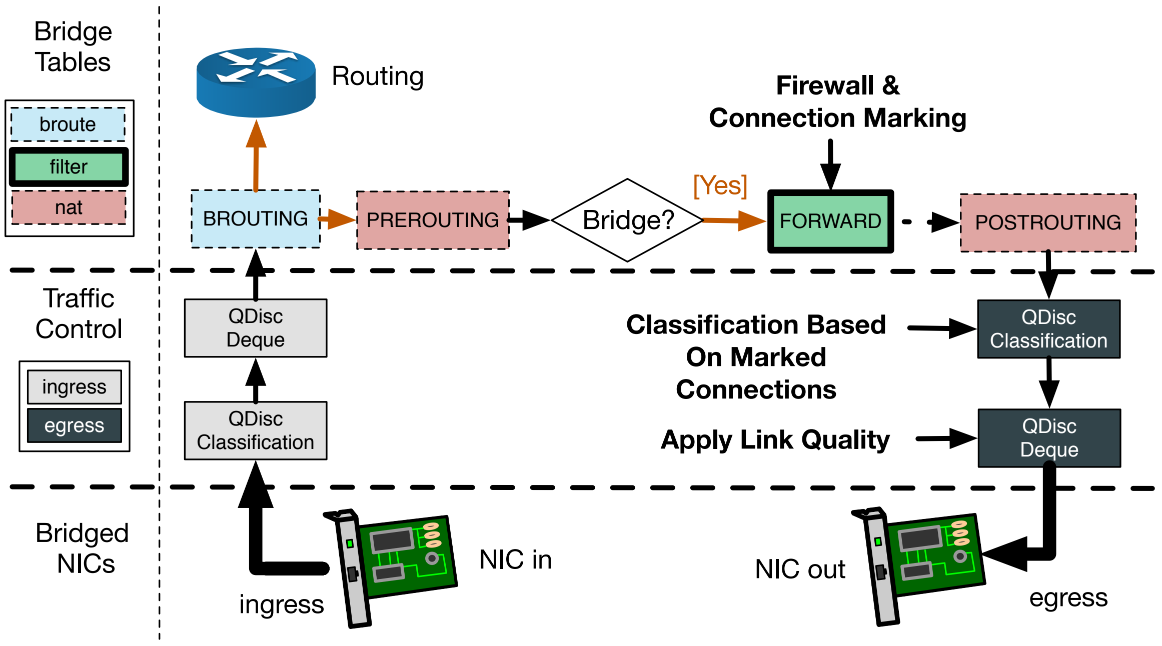


Figure 5.7: Ebtables & TC Interaction

only the delay is deployed in the *netem* capable link quality models[19]. *Netem* is classless, therefore the classful HTB has to be created first to create the hierarchy. The *netem* command introduces a delay of 16 ms ± 1,6ms where the delay depends to 25% on the last one. Traffic is classified by a filter and redirected to the appropriate class. Since the example is for the connection between node 1 and node 2, the traffic on interface *tap_00002_1*, where the connection is marked by *ebtables* with 2 (the number of connection), is redirected to class *1:2*.

```
# ebtables connection marking
ebtables --concurrent  -I wifi1 -i tap_00001_1 -o tap_00002_1 -j mark
 ↪  --set-mark 1 --mark-target ACCEPT
ebtables --concurrent  -I wifi1 -i tap_00002_1 -o tap_00001_1 -j mark
 ↪  --set-mark 1 --mark-target ACCEPT

# tc setup
tc -batch - <<<"
# htb root qdisc
qdisc replace dev tap_00001_1 root handle 1:0 htb
qdisc replace dev tap_00002_1 root handle 1:0 htb
qdisc replace dev tap_00003_1 root handle 1:0 htb

# set htb class
class replace dev tap_00001_1 parent 1:0 classid 1:1 htb rate 13500.0kbit
qdisc replace dev tap_00001_1 parent 1:1 handle 10: netem delay 4.00ms
 ↪  0.40ms 25%
```

[19]These are *LinkQualityModelWiFiLinear* and *LinkQualityModelWiFiExponential*.

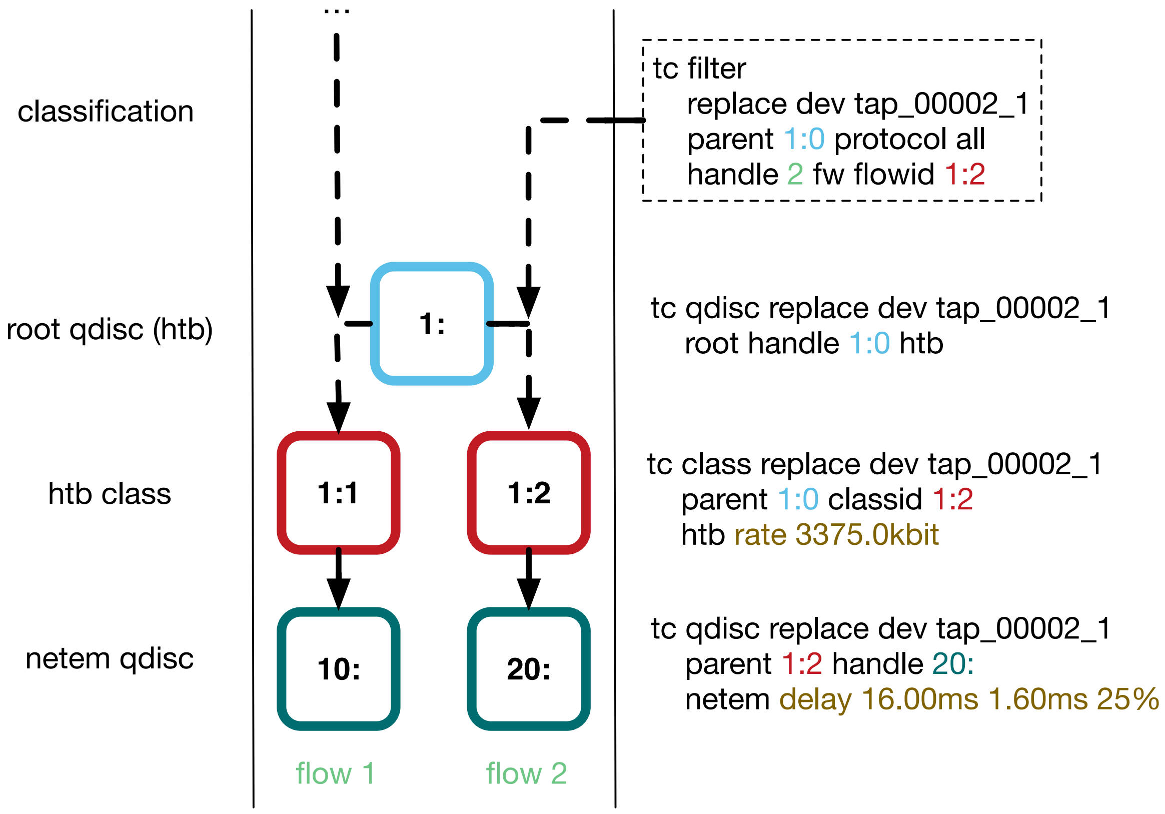


Figure 5.8: Node 2 (2 Connections) TC Setup

```
class replace dev tap_00002_1 parent 1:0 classid 1:1 htb rate 13500.0kbit
qdisc replace dev tap_00002_1 parent 1:1 handle 10: netem delay 4.00ms
↪  0.40ms 25%
class replace dev tap_00002_1 parent 1:0 classid 1:2 htb rate 3375.0kbit
qdisc replace dev tap_00002_1 parent 1:2 handle 20: netem delay 16.00ms
↪  1.60ms 25%
class replace dev tap_00003_1 parent 1:0 classid 1:2 htb rate 3375.0kbit
qdisc replace dev tap_00003_1 parent 1:2 handle 20: netem delay 16.00ms
↪  1.60ms 25%

# apply link quality settings to wifi connection
filter replace dev tap_00001_1 parent 1:0 protocol all handle 1 fw flowid
↪  1:1
filter replace dev tap_00002_1 parent 1:0 protocol all handle 1 fw flowid
↪  1:1
filter replace dev tap_00002_1 parent 1:0 protocol all handle 2 fw flowid
↪  1:2
filter replace dev tap_00003_1 parent 1:0 protocol all handle 2 fw flowid
↪  1:2"
```

Listing 15: Linux Ebtables and TC Commands Link Quality Model WiFi

All commands required to setup the link impairment in the *Chain 3* scenario with the *LinkQualityModelWiFiExponential* model are depicted in Listing 15.
The setup for the **Bridged LAN** network backend is much easier, since interfaces represent a connection. Therefore, no filters are required. Instead, the QDiscs can be directly attached to the interfaces.
Listing 16 outlines the required commands.

```
1 tc -batch - <<<"qdisc replace
2 dev tap_00001_1 root handle 1:0 htb default 1
3 qdisc replace dev tap_00002_1 root handle 1:0 htb default 1
4 qdisc replace dev tap_00002_2 root handle 1:0 htb default 1
5 qdisc replace dev tap_00003_1 root handle 1:0 htb default 1
6 class replace dev tap_00001_1 parent 1:0 classid 1:1 htb rate 13500.0kbit
7 class replace dev tap_00002_1 parent 1:0 classid 1:1 htb rate 13500.0kbit
8 class replace dev tap_00002_2 parent 1:0 classid 1:1 htb rate 3375.0kbit
9 class replace dev tap_00003_1 parent 1:0 classid 1:1 htb rate 3375.0kbit"
```

Listing 16: Ebtables and TC Commands: Link Quality Model WiFi

There is only one class for each interface. Therefore, packets are redirect to this class by default (command line switch: *default 1*, lines 3-5). Note that all *netem* capable link quality models can be used in combination with the *Bridged LAN* network backend.

5.5.3 WiFi

The *WiFi* network backend is a fork of *wmediumd*[20]. It registers with the *mac80211_hwsim* driver via a *netlink* socket and provides callbacks to receive frames from it. It simulates 802.11 QoS and MAC layer effects. After the simulation, the frames are sent back via the netlink socket to the driver. Since the bandwidth was not very high (around 2 Mbit/s), the wireless medium simulation has been removed. Further modifications were required to enable VMs to communicate with each other. Moreover, profiling has been done to improve performance.
Figure 5.9 outlines the control flow within the user-space application. There are two callbacks methods: *process_messages_cb* receives frames from the *mac80211_hwsim* driver and *nl_error_cb* is called for errors. The frames are then converted to a custom format and passed sequentially to two methods: *deliver_frame* sends the frame back to the kernel and notifies about the transmit status while of locally originated frames *send_frame* sends the frame via multicast to all nodes which subscribed to the multicast group.
These frames (expect locally originated ones) are then received by other nodes via the *process_incoming_frames* method which lives inside a thread. Received frames are passed to *deliver_frame* which sends the frames to the kernel.

Listing 17 shows the data structure used for sending 802.11 frames accross the network.

[20] `https://github.com/bcopeland/wmediumd`

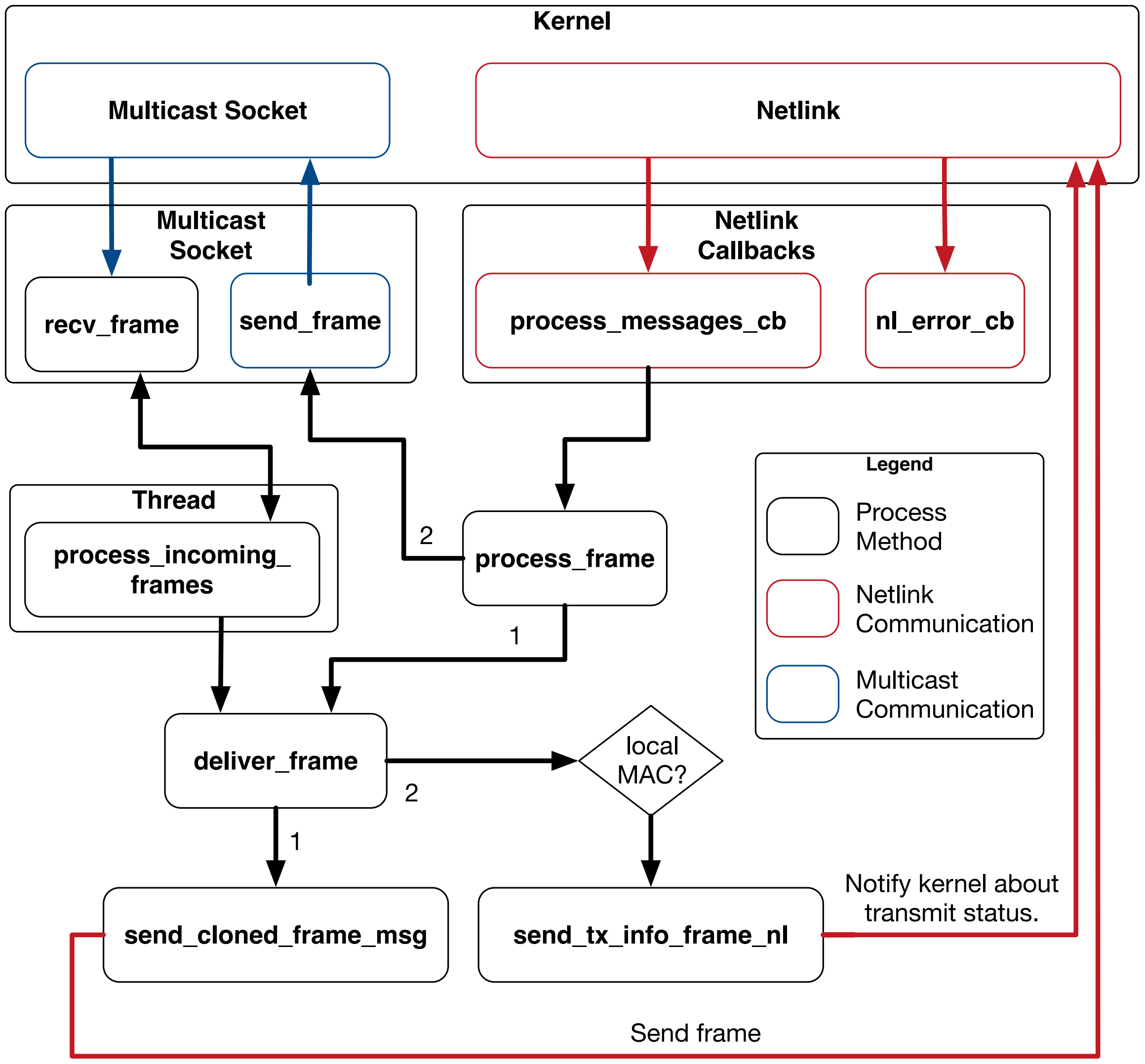


Figure 5.9: WiFi Network Backend Interaction

```
struct frame {
  uint16_t frame_len;

  // hwsim module related
  u64 cookie;
  uint32_t flags;
  uint32_t signal;
  uint32_t tx_rates_count;
  struct hwsim_tx_rate tx_rates[IEEE80211_TX_MAX_RATES];

  uint8_t sender;
  uint32_t data_len;
  // actual frame
  u8 data[0];
};
```

Listing 17: Frame Format

The attributes are mostly related to mac80211_hwsim. The actual 802.11 frame is located in the *data* field. A *sender* identifies a struct used internally for stations to represent *MAC* addresses etc.
The callback method for frame reception from the kernel is illustrated in Listing 18.

```
static int process_messages_cb(struct nl_msg *msg) {

  struct nlattr *attrs[HWSIM_ATTR_MAX+1];

  /* netlink header */
  struct nlmsghdr *nlh = nlmsg_hdr(msg);
  /* generic netlink header*/
  struct genlmsghdr *gnlh = nlmsg_data(nlh);

  struct station *sender;
  struct frame *frame;
  struct ieee80211_hdr *hdr;
  u8 *src;

  // handle only command frames
  if (gnlh->cmd == HWSIM_CMD_FRAME) {
    // parse into attrs
    genlmsg_parse(nlh, 0, attrs, HWSIM_ATTR_MAX, NULL);

    if (attrs[HWSIM_ATTR_ADDR_TRANSMITTER]) {

      u8 *hwaddr = (u8 *) nla_data(attrs[HWSIM_ATTR_ADDR_TRANSMITTER]);
      // get length
      unsigned int data_len = nla_len(attrs[HWSIM_ATTR_FRAME]);

      // allocate frame
      frame = mymalloc_zero(sizeof(*frame) + data_len);

      char *data = (char *) nla_data(attrs[HWSIM_ATTR_FRAME]);
      memcpy(frame->data, data, data_len);
      frame->data_len = data_len;

      // ...

      frame->sender = sender->index;
      frame->frame_len = sizeof(struct frame) + frame->data_len;

      process_frame(frame);
39      }
40    }
41  }
```

Listing 18: Netlink Callback

It basically reads attributes from the netlink message and stores them in the custom frame format *struct frame*. The frame is then passed to *process_frame* (line 38) which handles the frame as described previously. Note that the *WiFi* network backend only works with machines having the same endianess since converting data to big endian reduces performance and is unnecessary because at the time of writing it is not possible to start VMs with different processor architectures.

5.6 Distributed

Distributed communication between a *coordinator* and clients is achieved with *ZeroMQ*. The server implementation is depicted in Listing 19. Both patterns (*Request-Reply* vs. *Publish-Subscribe*) use at least 2 sockets (lines 7-10, 12-16).

```
1   class ZeroMQServer(StartableObject):
2
3     def __init__(self):
4
5       self.context = zmq.Context.instance(CNT_ZMQ_THREADS)
6
7       # create the router socket
8       self.router_socket = self.context.socket(zmq.ROUTER)
9       addr = "tcp://*:{}".format(Protocol.PORT_DEFAULT_SERVICE)
10      self.router_socket.bind(addr)
11
12      # create reset socket
13      self.reset_socket = self.context.socket(zmq.PUB)
14      self.reset_socket.setsockopt(zmq.IDENTITY, bytes(random.randrange(1,
    ↪   sys.maxint)))
15      addr = "tcp://*:{}".format(Protocol.PORT_PUB_RESET_SERVICE)
16      self.reset_socket.bind(addr)
17
18      # protocol
19      self.protocol = Protocol.factory()()
20
21      # events
22      self.wait_for_scenario_config = threading.Event()
23      self.wait_for_nodes_started = threading.Event()
24
25      def _start(self, cnt_peers):
26          self.cnt_peers = cnt_peers
```

```
27
28          self.handle_state_register()
29
30          # wait until the scenario config is set, because the tunnel
  ↪   addresses
31          # (received in the next step) are written to it
32          log.info("waiting until scenario config is set ...")
33          self.wait_for_scenario_config.wait()
34          self.handle_state_information_exchange()
35          self.handle_state_start_nodes()
```

Listing 19: ZeroMQ Server

A router socket is the asynchronous implementation of the *Request-Reply* pattern on the server side (line 7-10). On the client side, a *Request* socket is created which allows communication only in an alternating sequence of *send* and recv calls. A *publish* socket is used by the *coordinator* to reset clients if an error occurred during simulation (lines 12-16, *Publish-Subscribe* pattern). A *protocol* handles the serialization/deserialization of data (line 19).
The *ZeroMQServer* class is started in an own thread which is responsible to execute the *_start* method. An extra thread is necessary since the *ZeroMQ* sockets are used in blocking-mode. The *_start* method depicts the states of the DFA. Each state is handled by an own method.

5.6.1 State Register

Initially, clients need to register with the *ZeroMQ* router socket (line 28). The state is part of every message between a client and its *coordinator*. The number of clients are defined in the *Global Config*, hence the *handle_state_register* method blocks until all clients are registered. The *Scenario Config* is required since tunnel addresses and the node placement on *clients* needs to be communicated to all clients. Hence, line 33 blocks until a user has set the *Scenario Config*. Clients get an ID assigned in the order they connected to the *coordinator*. The ID is included by clients for requests in further states.

5.6.2 State Echange

In this state (line 34), the client informs the *coordinator* about the IP which shall be used for tunnels and a score dictionary. The scoring is implemented by the class *ServerScore*. The amount of free RAM is read with the help of the *psutil* library. The *bogomips* value is read from */proc/cpuinfo*. The score is used as input for the *NodeDistributionStrategy* for node scheduling.

5.6.3 State Start Nodes

In this state all clients start the VMs and sync with the *coordinator*. After the nodes have been started, the *wait_for_nodes_started* event is set (declared in line 23).

5.6.4 State Distance Matrix

The method *handle_state_distance_matrix* is called by the *SimulationManager* first, if the event has been set. Either the *RunLoop* thread performs steps in a predefined time unit, or the user may step via the *RPC* interface. Both end up in a call to *handle_state_distance_matrix* which calls the method depicted in Listing 20.

```
1  class ZeroMQServerRouter(ZeroMQServer):
2
3    def _handle_state_distance_matrix(self, distance_matrix_per_server):
4
5      expect_distance_matrix =
   ↪   self.get_expecter_state(States.STATE_DISTANCE_MATRIX, 1)
6      # sync clients and send each his distance matrix
7      ResponderPerServerID(self.router_socket, self.protocol,
   ↪   expect_distance_matrix, distance_matrix_per_server)()
```

Listing 20: ZeroMQ Server Router Distance Matrix

The Listing outlines the implementation of the *Request-Reply* pattern for the distribution of the distance matrix by the *coordinator*. Moreover, it shows the implementation of the DFA. The variable *distance_matrix_per_server* holds the distance matrix for each client ID. This may be either the full distance matrix, or only the relevant part of the distance matrix for each client. The communication of the *coordinator* with its clients is abstracted by two classes: The *Expecter* class hides the asynchronous communication. For each client, a multipart message with n arguments is expected. Each argument is deserialized separately. Moreover, the state and the number of arguments of a client message is checked. The *Expecter* class which is used for the *Distance Matrix* state is shown in line 5. Only the ID is required from clients in this state.
A *Responder* class handles the replies to the clients. Either each clients gets the same reply (*ResponderArgument*) or there is a different reply for each client (*ResponderPerServerID*). The *ResponderPerServerID* is used in line 7 to send each client the appropriate distance matrix. The implementation of the *Publish-Subscribe* distance matrix distribution is depicted in Listing 21.

```
class ZeroMQCServerPubSub(ZeroMQServer):

  def _handle_state_distance_matrix(self, distance_matrix):
    self.sync_subscribers()
    self.pub_socket.send( distance_matrix )

  def sync_subscribers(self):
    expect_distance_matrix =
↪   self.get_expecter_state(States.STATE_DISTANCE_MATRIX, 1)
    ResponderArgument(self.router_socket, self.protocol,
↪   expect_distance_matrix, b'')()
```

Listing 21: ZeroMQ Server Publish Distance Matrix

In the *Request-Reply* implementation the clients have been synchronized via the router socket, since a clients recv() call waits for a response of the *coordinator*. In the *Publish-Subscribe* pattern in contrast, communication is unidirectional. Hence, the router socket is used for synchronization (lines 7-9) while the distance matrix is sent via the *Publish* socket (line 5).

5.6.5 Distance Matrix Mapping

The mapping of the distance matrix to clients is depicted in Listing 22.

```
1  def map_distance_matrix_to_servers(self, distance_matrix):
2    res = defaultdict(dict)
3
4    if not config.is_publish_individual_distance_matrices():
5        for server_id in scenario_config.get_distributed_server_ids():
6          res[server_id] = distance_matrix
7        return res
8    else:
9        # create a dict for each server
10       for server_id in scenario_config.get_distributed_server_ids():
11         res[server_id]
12
13       for (x, y), distance in distance_matrix.items():
14         res[self.get_server_for_node(x)][(x, y)] = distance
15         res[self.get_server_for_node(y)][(x, y)] = distance
16
17       return res
```

Listing 22: Distance Matrix To Node Mapping

The method *config.is_publish_individual_distance_matrices()* (line 4) reflects a config value in the *Global Config*. If no individual distance matrices shall be created, for each *server_id* the distance matrix is stored (lines 4-5). In the other branch, the node placement matrix is used to filter out the entries of the distance matrix for a client. All nodes which are hosted by a client are relevant to it (lines 8-15). Which client is responsible for which node is determined with the *get_server_for_node* method (lines 14-15).

5.7 VM Preparation

The following illustrates how VM images can be created and deployed. Moreover, it points out the required modifications of a VM such that it works with MiniWorld.
New images can be created with the commands shown in Listing 23.

```
1 wget http://saimei.acc.umu.se/debian-cd/8.6.0/amd64/iso-cd/
    ↪ debian-8.6.0-amd64-netinst.iso
2 qemu-img create -f qcow2 debian_8.qcow2 5G
3 kvm -boot once=dc
4 -vga qxl -spice port=5900,addr=127.0.0.1,disable-ticketing
5 -redir :<host_port>::22
6 -cdrom debian-8.6.0-amd64-netinst.iso debian_8.qcow2
```

Listing 23: Image Deployment

First, an image has to be downloaded. Debian 8 (Jessie) is used in the example (line 1). Then a *QCOW2* image is created which serves as the hard disk for the VM. The VM is booted from the live image (line 2). The user can then install the OS to the harddisk of the VM. Note that starting KVM without the *-vga* switch does not work for images which have a graphical installer. The UI can be accessed with *spice* compatible programs [21].
After installing the VM, it can be started without the live image by leaving out the *–boot* and the *-cdrom* command line switches. The *-redir* CLI switch redirects the port 22 to localhost. This enables accessing the VM via *ssh* if the network is configured by means of Dynamic Host Configuration Protocol (DHCP) in the VM. Moreover, the VM can access the internet for further deployment. Note that ICMP does not work with the user network backend (*SLIRP*) of Qemu.
The modifications of the VM required by MiniWorld depend on the boot loader and the init system. For systems with *grub*, the serial console can be enabled by modifying the */etc/default/grub config file*. Listing 24 enables the serial console and disables the new NIC naming scheme for Ubuntu 16.04 systems.

```
GRUB_CMDLINE_LINUX="console=tty1 console=ttyS0 net.ifnames=0 biosdevname=0"
GRUB_TIMEOUT=0
GRUB_TERMINAL=console
```

Listing 24: Grub Modifications at /etc/default/grub

This allows the *NetworkConfigurator* to configure based on the *ethX* naming scheme. The timeout is not required, but improves VM boot times.
The modification of *grub*[22] redirects the kernel boot log to the serial console so that MiniWorld can detect when the boot process is over.
There is no autologin mechanism implemented. Therefore, the root user is expected to be logged in on the serial console's shell. The modification depends on the init system. Ubuntu 16.04 uses *Systemd* while older versions used the *Upstart* init system. The modifications for both systems are depicted in Listing 25 and 26 respectively.
Another reduction of VM boot times can be achieved by disabling any DHCP configuration.

[21] Linux: apt-get install libvirt-bin. Mac: brew cask instal remoteviewer. Run *remote-viewer spice://0.0.0.0:5900*
[22] The command *update-grub* has to be run after the file is modified.

```
mkdir /etc/systemd/system/serial-getty@.service.d
cat << EOF >
/etc/systemd/system/serial-getty@.service.d/override.conf
[Service]
ExecStart=
ExecStart=-/sbin/agetty --keep-baud 115200,38400,9600 -a root %I $TERM
EOF
```

Listing 25: Systemd Modifications

```
T0:23:respawn:/sbin/getty -L ttyS0 --autologin root 38400 vt100
```

Listing 26: Sysvinit Modifications

5.8 CLI

In the following, the CLI is presented. Initially, the event system is presented in Section 5.8.1 since it provides the the CLI with the current progress of operations. The CLI implementation and error checking is discussed in Section 5.8.2. Finally, an experiment is depicted which shows the CLI API in the distributed mode (Section 5.8.3).

5.8.1 Event System

Monitoring the progress of operations in MiniWorld is achieved through a central system called *Event System*. It defines several events on a per node basis. The complete progress for an event is then the progress of all nodes for this particular event. Progress calculations may not be so correct that they hit 100% at the end. Therefore, extra context managers help to ensure that while leaving the current context, the progress of an event is updated to 100%. Listing 27 outlines the usage of the *Event System*.

```
>>> es = EventSystem(["event_boot"])
>>> # instead of constructor: es.events.add(es.EVENT_VM_BOOT)
>>>
>>> with es.event_init("event_boot") as event_boot:
>>> for i in range(5):
>>>     time.sleep(0.2)
>>>     event_boot.update([1], 0.2 * i)
>>>     event_boot.update([2], 0.1 * i)
>>>     print "avg: %s" % dict(es.get_progress())
>>>     print "per node: %s" % pformat(es)
# output for i==1
```

```
12 avg: {'vm_boot': 0.15000000000000002, 'total_progress':
    ↪ 0.15000000000000002}
13 per node: OrderedDict([(1, OrderedDict([('vm_boot', 0.2)])), (2,
    ↪ OrderedDict([('vm_boot', 0.1)]))])
14 >>> # progress is not 100% yet
15 >>> time.sleep(5)
16 >>> # but after the context manager exits
17 >>> print "finishing ..."
18 >>> print es.get_progress()
19 >>> pprint(es)
20 [('total_progress', 1.0), ('vm_boot', 1.0)]
21 OrderedDict([(1, OrderedDict([('vm_boot', 1.0)])), (2,
    ↪ OrderedDict([('vm_boot', 1.0)]))])
```

Listing 27: Event System Demo

The system is initialized with one event (*event_boot* (line 1)). Note that events can also be added later. The context manager *event_init* sets the progress to WiFi at the beginning of the context/block. The *event_boot* is an *Event* object which can be used to update or finish the progress. The example updates the progress for two nodes independently. The progress is 0.2 for node one and 0.1 for node for i=1 (line 13). The average progress is calculated from both nodes (line 12). After the context is left, the progress is updated to 1.0 (shown in lines 20-21). There are cases where the progress is not controlled from inside a loop. For this case, it is possible to define for which node ids the progress has to be set to 1.0 after leaving the context. The same applies for initializing the progress for an event with the context manager. Moreover, the total progress may not be known in all cases. Therefore, it is possible to add a float value instead of updating the complete progress.
Figure 5.10 depicts the currently deployed events and their order. First a VM is booted (1). Afterwards the serial console's shell is entered (2). Then the *Pre Network Shell Commands* are executed (3). The *Post Network Shell Commands* (5) are first executed if the network backend has been set up (4). Finally the network is setup in terms of IP provisioning (6) and the connectivity is checked (7). For event number 6, the number of connections that have to be established are calculated by the *NetworkManager*. Note that even though the event update for a node is sequential in terms of events, it is not for the whole progress, since nodes are started and managed in parallel. At the end of the next Section, a CLI view of the progress is shown which is based on the *EventSystem*.

5.8.2 CLI & Error Checking

The CLI is built on top of the RPC interface. The usage of the RPC interface allows different clients and technologies to control MiniWorld. A web UI might follow in future releases of MiniWorld.
The CLI is provided by the *mw.py* script. It uses the *argparse* module to implement the CLI and provides a subparser for each command. A parser can be associated with a function. The function of the parsed command can then be accessed via the parsed arguments. This function is linked internally to an *ActionClass*. By means of a decorator, the access to the class is provided in the *self* argument to the function. Listing 28 outlines a method used

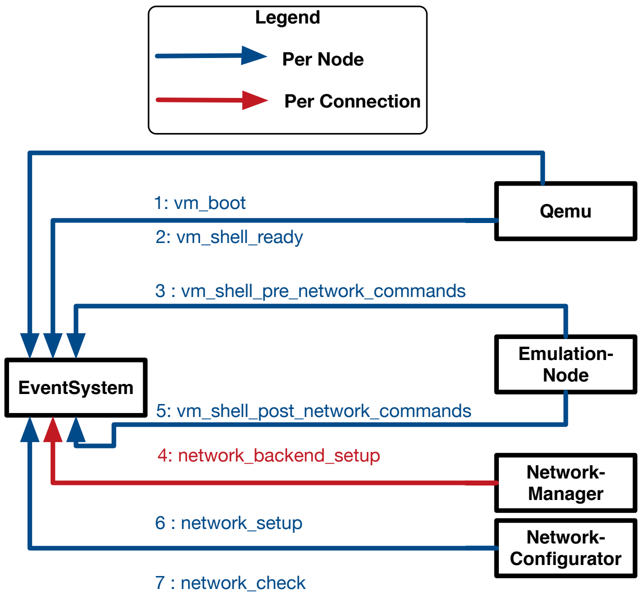


Figure 5.10: Event System Interaction

for the *./mw.py step* command. The variable *args* contains all parsed information from the parser (*argparse Namespace object*).

```python
@new_action_decorator
def action_step(self, args):
  return self.connection.simulation_step(args.steps)
```

Listing 28: CLI Action Method

RPC connection establishment, error checking and RPC server switching in the distributed mode is abstracted by the *ActionClass*, which is depicted in Listing 28. The *__call__* method establishes a connection to the RPC server (line 7).

```python
1  class Action(object):
2
3    def __call__(self, args):
4
5      self.addr = RPCUtil.addr_from_ip(args.addr, RPCUtil.PORT_COORDINATOR)
6      log.info("connecting to rpc server '%s'", self.addr)
7      self.connection = self.get_connection(self.addr)
8      if not args.no_check:
9        # check for errors first
10       self.connection.simulation_encountered_exception()
11
12     # switch to the rpc server which holds the node
```

```
13    if hasattr(args, "node_id") and args.node_id is not None:
14      self.connection = self.get_connection_server(node_id=args.node_id)
15
16   def __new__(cls, *args, **kwargs):
17
18    cls._instance = super(Action, cls).__new__(cls)
19    return cls._instance(*args, **kwargs)
20
21   def get_connection_server(self, node_id=None, server_id=None):
22
23    if config.is_mode_distributed():
24
25      # ask local rpc for ip of the server which holds the node
26
27      if server_id is None and node_id is not None:
28        server_id = self.connection.get_server_for_node(node_id)
29      else:
30        server_id = server_id
31
32      print >> sys.stderr, "using server: %s" % server_id
33      addr =
   ↪ self.connection.get_distributed_address_mapping()[str(server_id)]
34      addr = RPCUtil.addr_from_ip(addr, RPCUtil.PORT_CLIENT)
35      log.info("switching to rpc server '%s'", addr)
36      # drop local connection and use remote server
37      connection = self.get_connection(addr)
38
39      # check for errors first
40      connection.simulation_encountered_exception()
41
42      return connection
43
44    # use local connection
45    return self.connection
```

Listing 29: CLI Action Class

Errors in MiniWorld are stored at a central place. This is used by a custom Thread class, to stop and store the exception at this place. Since there is no way to directly tell the client that an error occurred (no standing connection), a client has to manually check for errors. The *__call__* method performs error checking for all CLI methods, before the actual RPC call is done (lines 8-10). Some actions are required[23] to be redirected to a specific server in the case of the distributed mode. The method *get_connection_server* handles this case. For that purpose the RPC interface of the coordinator (line 33) is used to find out the server which manages the specific node id (line 28). Afterwards, the address of the RPC server

[23]If *node_id* is in the parser namespace object.

is figured out and finally the connection is switched to the other RPC server (line 33, 37). Error checking is also performed for the new RPC server (line 40).
The *new_action_decorator* shown in Listing 30 ties a method together with the *ActionSubClass*.

```
1  def new_action_decorator(fun):
2
3    class ActionSubClass(Action):
4
5      def __call__(self, args):
6        super(ActionSubClass, self).__call__(args)
7
8        # supply ActionSubClass to the method
9        return fun(self, args)
10
11   return ActionSubClass
```

Listing 30: CLI Action Decorator

It does so by providing a decorator for methods. This exchanges the actual method with the *ActionSubClass* depicted in lines 3-11. It mimics a method call with the *__call__* method. Methods require access to the connection object of the *ActionClass*, therefore self is supplied to the function in line 9.
Furthermore, the CLI points out the usage of the *EventSystem*. The progress for each event is summed up and serialized as a dictionary. This is done periodically to update the progress on the CLI. For that purpose, the starting of a RPC scenario is set to be non-blocking so that the RPC connection can be used to fetch the progress periodically. Listing 31 shows the visualization of the progress on the Command Line Interface.

```
Scenario starting: |

Overall Progress         :  59% |>>>>>>>>       ||      <<<<<<<<|
vm_boot                  : 100% |||||||||||||||||||||||||||||||||
vm_shell_ready           : 100% |||||||||||||||||||||||||||||||||
vm_shell_pre_network_comm:  96% |------------------------------ |
network_backend_setup    :   0% |                               |
vm_shell_post_network_com:   0% |                               |
network_setup            :   0% |                               |
network_check            :   0% |                               |
```

Listing 31: CLI Progress View

5.8.3 MiniWorld API Demonstration

```
#!/usr/bin/env bash

hosts=(root@androbox57 root@androbox58 root@androbox59 root@androbox60
 ↪ root@androcloud root@rechenschieber root@nicerboxbig)
port=2222

function clean_up {...}
trap clean_up EXIT

hosts_str=‘printf "%s:$port," "${hosts[@]}"‘
hosts_str=${hosts_str::-1}
experiment_dir=experiments/results/distributed_chain_256
peer_stats=$experiment_dir/peers.txt
cnt_nodes=256
rm $peer_stats
mkdir -p $experiment_dir

set -e

echo "starting cpu sensor on all machines"
fab cstart_cpu_sensor -P -H $hosts_str &
fab_pid=$$

echo "starting scenario ..."
./mw.py start MiniWorld_Scenarios/experiments/distributed/
distributed_chain_256_serval.json

echo "setup network"
./mw.py step

echo "starting serval on all nodes ..."
./mw.py exec "servald start"

cnt_peers=0
until [ $cnt_peers -eq $cnt_nodes ]; do
    cnt_peers=$(./mw.py exec --node-id 1 'servald id peers' 2>/dev/null |
 ↪  echo $((‘wc -l‘-3)))
    time=$(date +%s)
    echo "$time,$cnt_peers" | tee -a $experiment_dir/peers.txt
    sleep 0.1
done
```

Listing 32: Use Case

The code depicted in Listing 32 shows a *Chain 256* network topology used for testing the

Serval Mesh Software[24]. It is used in Section 6.5.6 to provide performance measurements of the distributed mode.
There is nothing special in the *Scenario Config*, therefore it is not shown. Bash has been chosen as scripting language because of its easy shell command execution. Of course, the MiniWorld CLI is accessible from every programming language. The experiment uses seven computers in the distributed mode. There is a cpu sensor started with fabric[25]. Moreover, the time which Serval requires to find all its neighbours is measured (lines 34-39).
In line 3 the participating hosts are declared for ssh automation via fabric. A cleanup method for the fabric processes and fetching the logs via scp is depicted lines 6-7. The host command line argument for fabric is build in lines 9-10 . The result directory and result file are defined in lines 11-12.
The number of Serval peers, node one is required to see, is defined in line 13.
First, the cpu sensor is started with fabric on all computers (lines 19-21). Then the scenario is started (lines 23-25). This starts the nodes in parallel on all computers.
Note that at this point, no network between the nodes is set up except the management network. The scenario is started without a *RunLoop*, therefore the user has to manually step. The first step (line 28) then sets up the network topology.
Afterwards *servald* is started on all nodes in the cluster. The CLI interface connects to the RPC interface which distributes the requests sequentially to all computers in the cluster via the RPC interface. Finally, the number of Serval peers are logged. For that purpose, the command *servald id peers* is periodically executed on node one (line 35). The request is again forwarded by the *Coordinator* to the appropriate computer which hosts the node/VM. The experiment exits if node one is able to see all of its peers.
The *clean_up* functions copies the logs via scp afterwards.

[24]`www.servalproject.org`
[25]Python SSH automation tool: http://www.fabfile.org

6 Evaluation

This section presents the experimental evaluation of MiniWorld. First the test environment is introduced in Section 6.1. Afterwards the start times of *OpenWrt Barrier Braker* and *Debian 8* VMs are examined. For that purpose, the different *Boot Modes* are compared in terms of start time and CPU overhead (Section 6.2). Following is the *Snapshot Start* mode which reduces VMs start times heavily (6.2.2).
Afterwards the different *Qemu NIC models* are evaluated in terms of bandwidth (Section 6.3). Network throughput, delays and switching times of the different network backends are examined in Section 6.4.
With the help of the *WiFi* network backend, a 802.11s network with 32 hops is build and evaluated in Section 6.4.3.
Last but not least, the distributed mode of MiniWorld is evaluated. The communication overhead is examined in Section 6.5.1. For further studies a new test system is introduced in Section 6.5.2. Following is the study of boot times (Section 6.5.3) and network switching times (Section 6.5.4). Moreover, the overhead of tunnels in terms of bandwidth and delay is examined (Section 6.5.5). Finally a *Chain 256* mesh network with *Serval*, running on 7 computers, in MiniWorld's distributed mode is presented in Section 6.5.6.
All experiments have been conducted with the *CORE Mobility Pattern*. The main bottleneck of experiments was neither Random Access Memory (RAM) nor network. Instead CPU showed to be the limiting factor. Therefore, heatmaps are used to show the CPU usage of the various experiments. Moreover, experiments are repeated 10 times to get statistical sound results.

6.1 Test Environment

All experiments where run using *Docker* containers. This enables the easy deployment of MiniWorld. Since *iproute2* has to be build according to the kernel version, there is no possibility to include the correct *iproute2* version in the *Docker* image. Although, there is an installation script which automates the *iproute2* deployment. The experiments except for the distributed mode, have been performed on a single computer. The technical details of it are depicted in Table 6.1. It has 16 physical CPU cores and 64 virtual cores. 256GiB of memory allow to run a lot of VMs concurrently. Throughout all experiments *iproute2* version 4.2.0 (Git[1]) and *Qemu* version 2.6.0 (Git[2]) has been used.

Table 6.2 shows the VMs which are used in the experimental evaluation. **OpenWrt Barrier Braker** has been compiled manually, hence only a minimum of kernel modules and software packages are built-in. The **Debian 8** image has been installed from a netinstall image where only the options *SSH Server* and *Standard system utilities* have been chosen in the software

[1]`https://kernel.googlesource.com/pub/scm/linux/kernel/git/shemminger/iproute2`

[2]`https://github.com/qemu/qemu`

[3]`https://github.com/svinota/pyroute2`. Commit ID: commit 13f1adb1ab2d5ca8927f1eb618bbb9317-0b61c33

Resource	Value
Host OS	Ubuntu 14.04.4 LTS
Docker OS	Ubuntu 14.04.4 LTS
Kernel	4.2.0-42-generic
RAM	32x DIMM DDR3 Synchronous 1600 MHz 8GiB
HDD	1862GiB, 15000rpm
CPU	AMD Opteron(tm) Processor 6376 @2.3GHz, Turbo 3.2GHz
CPU Cores Physical/Virtual	16/64
Network	NetXtreme II BCM5709 Gigabit Ethernet 1Gbit/s
Iproute2	Git release v4.2.0
Pyroute	Git [3]
Qemu	Git release v2.6.0

Table 6.1: Technical Data Test System 1

Image Name	Size (MiB)
OpenWrt Barrier Braker	68
Debian 8	1568

Table 6.2: Node Images

selection dialog.
While the custom built *OpenWrt Barrier Braker* image has only a size of 68 MiB, the *Debian 8* VM is far bigger with a size of 1568 MiB. The images do not only differ in disk image size, they also show different memory consumptions after boot: *OpenWrt Barrier Braker* only needs 10MiB whereas *Debian 8* VM requires 78MiB of RAM[4].

6.2 QEMU

In this Section, the boot times of Qemu VMs are evaluated. The first experiment evaluates the performance of the different *Boot Modes*. Figure 6.1 outlines the results of several VM start modes. The number of nodes has been increased exponentially. The image used for all nodes is *OpenWrt Barrier Braker* with 32 MiB of memory. Neither *Pre Network Shell Commands* nor *Post Network Shell Commands* are used in the appropriate *Scenario Configs*. The abbreviations presented in the Figure are as follows: Boot Prompt (BP), Shell Prompt (SP) and Ram Disk (RD). The Ram Disk tries to leverage the huge amount of memory which the test system offers. For that purpose a *ramfs*[5] file system is mounted on */tmp/MiniWorld* where all MiniWorld files are located. In contrast to the *tmpfs* file system, *ramfs* prevents swapping. If swapping occurs, the number of VMs should be reduced. Since MiniWorld runs inside a container, memory limits can prevent the container from freezing the host.
The number of nodes have been doubled until a total number of 512 VMs has been reached. The actual limit of the test system for starting VMs is between 512 and 1024. Starting 1024 VMs was no problem, but creating snapshots for all of them in memory (due to the ram

[4]Started with -m 2048M and executed command *cat /proc/meminfo*.

[5]`https://www.kernel.org/doc/Documentation/filesystems/ramfs-rootfs-initramfs.txt`. Last viewed on 01.12.2016

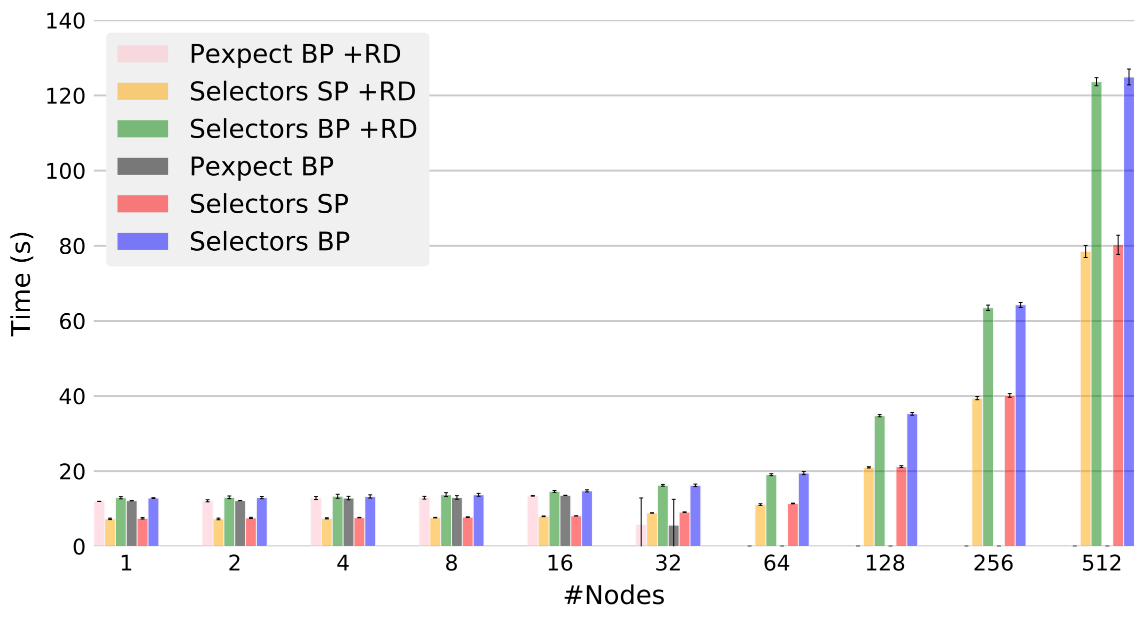


Figure 6.1: Boot Mode Times OpenWrt Barrier Braker, 32MiB RAM

disk) was.
The VMs were started concurrently. The test machine offers 64 virtual cores, thus there are nearly no performance differences between starting 1 or 64 nodes. Doubling the number of nodes to 64 however results in approximately doubling the time required to start all VMs. Figure 6.1 points out a problem for the *Pexpect Boot Mode*. Since it uses *select* system call to check for I/O, the number of file handles was too high for *pexpect*, hence starting 32 VMs was not possible for some runs. The Ram Disk feature shows only an improvement of 1-2 seconds. The reason for this might be that each node uses the same backing image, thus it is already cached. During boot probably not very much disk write operations are performed. Further studies are required to evaluate if the *Ram Disk* feature improves performance in a more advanced emulation where nodes are not only booted but some software is running over a long time. Write operations are all done to memory, since the write layers are located in the Ram Disk too.
The experiment shows that MiniWorld does not introduce a bottleneck while starting and configuring nodes, at least for the tested number of VMs. The *Selectors SP* mode is always faster than *Selectors BP*. The explanation for this is that the *Shell Prompt* appears before the *Boot Prompt*, at least for the tested *OpenWrt* version. Moreover, simulating pressing enter is necessary for the *Shell Prompt* since it does not appear otherwise. The *Boot Prompt procd: - init complete* - has not been inserted manually. It is part of the *OpenWrt Barrier Breaker* boot process output. However, more recent versions seem to miss it.
The posed CPU resources of the two *Selectors* boot modes are depicted in Figure 6.2. At the colorbar on the ride sight of each diagram, one can see that the maximium CPU value is higher for the *Selector Boot Prompt* mode. The reason for this is the difference in the implementation. The *Selector Boot Prompt* reads the whole boot output and tries to match the *Boot Prompt* as regular expression against it for each new outcome of the UDS. This is necessary since the *Boot Prompt* may be split up between two socket read() calls. Contrary to this, the *Selector Boot Prompt* tries to enter the shell in predefined time intervals and gets

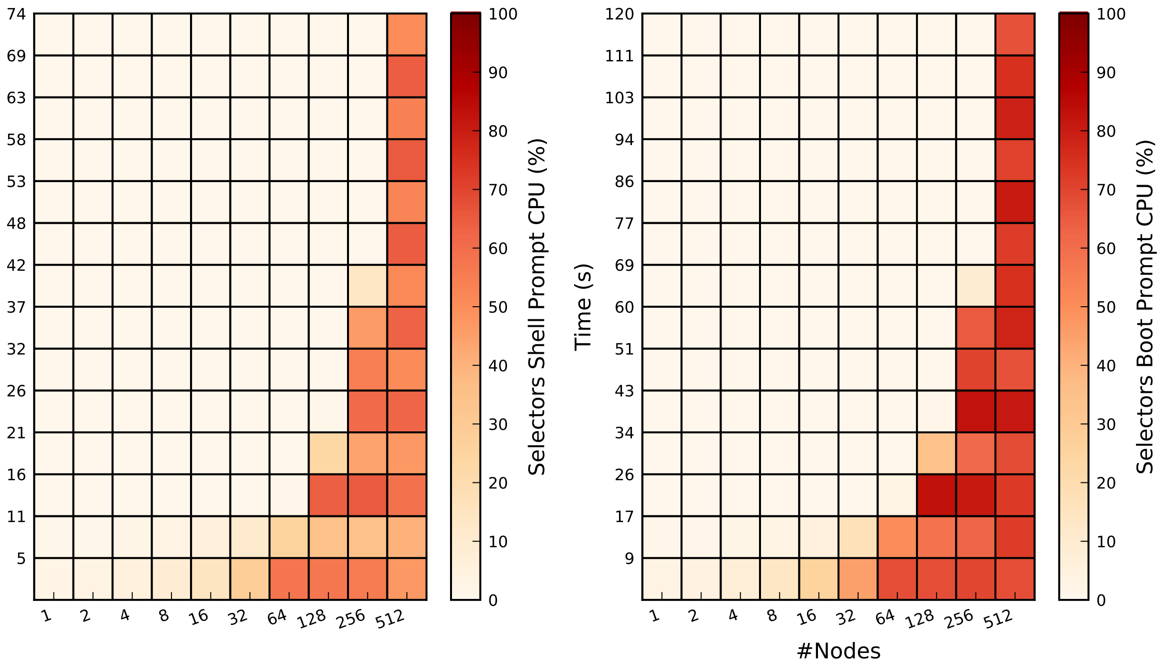


Figure 6.2: Boot Modes CPU, OpenWrt Barrier Braker, 32MiB RAM

the *Shell Prompt* for every simulated enter if the VM is ready. Therefore, no output has to be stored.
Using the *Shell Prompt* to detect when a VM is started is easier but the VM may not be ready yet. For example an interface may not be ready and commands to configure the NIC by MiniWorld may fail. Therefore, one has to evaluate for each image if the *Selectors Shell Prompt* mode leads to problems. The *Selectors Boot Prompt* mode is preferable but may require to insert a *Boot Prompt* into the boot process of the VM. Note that the *Shell Prompt* is mandatory since it is needed for the *REPLable* mechanism.

6.2.1 Image Comparison

Another experiments aims at pointing out the differences between the node images. Both have been started with the *Selectors Shell Prompt* boot mode and no shell commands are sent to the VMs serial console. The *Debian 8* VM required 256MiB of RAM instead of 32MiB (*OpenWrt*). Figure 6.3 shows the required boot time for each image. Moreover, it points out that *OpenWrt* is approximately twice as fast as *Debian*. Since the *Shell Prompt* boot mode has been used, the VMs may not be fully booted, but offered a shell prompt. Moreover, doubling the number of nodes by factor 64 (number of virtual CPU cores) doubles the start times for each image. Therefore, MiniWorld scales **linear**. The CPU overhead introduced by each image is illustrated in the heatmaps in Figure 6.4. It points out that *Debian 8* poses higher requirements on the CPU.
The Ram Disk feature showed again small improvement in boot times.
The selection of the best fitting image depends on the requirements a user has: OpenWrt is very fast but is harder to configure, especially if a custom image needs to be built from scratch. Debian on the other side is easy to install and deploy but reduces performance as the experiment pointed out.

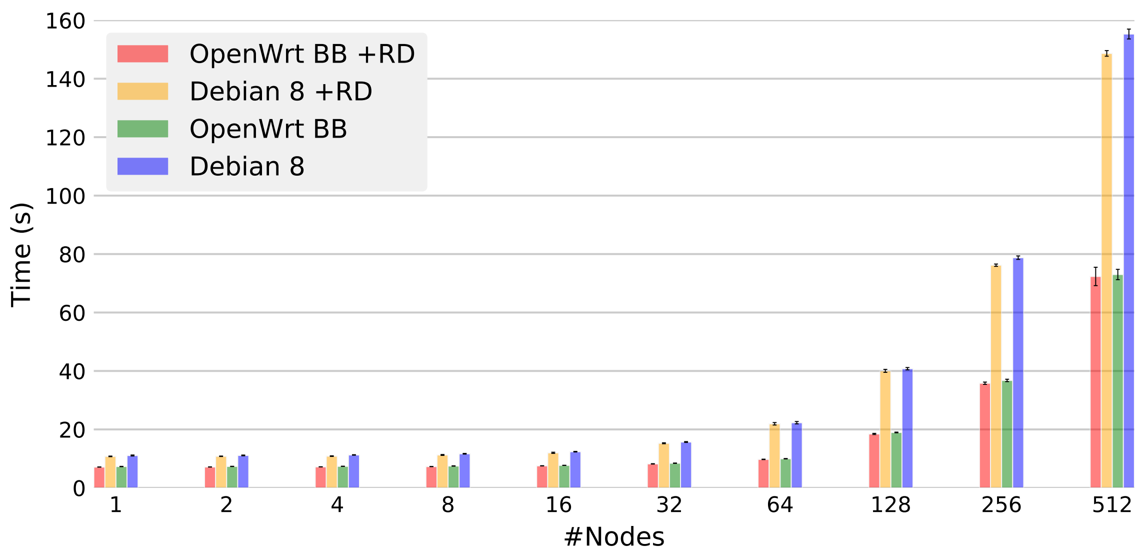


Figure 6.3: OpenWrt vs. Debian 8 Boot Times. OpenWrt RAM: 32MiB. Debian RAM: 256MiB. Boot Mode: *Selectors Shell Prompt*

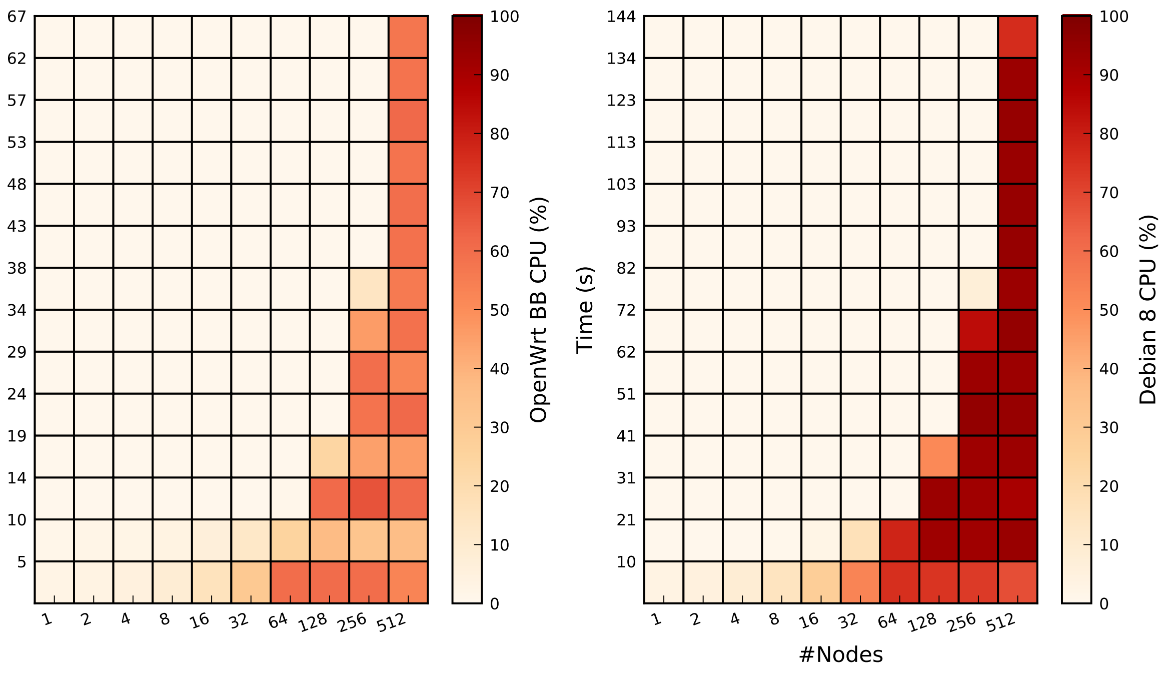


Figure 6.4: Boot Times CPU Usage (+RD): OpenWrt Barrier Braker vs. Debian 8

6.2.2 Snapshot Boot

The experiments in the last Section showed the overhead which is introduced by full system virtualization. Since experiments have to be run multiple times with the same configuration to get statistically sound results, VM snapshots can be used to improve VM boot times drastically. Snapshots are taken after the VM has been started the first time. The *Pre Network Shell Commands* are incorporated into the snapshot, the *Post Network Shell Commands* are not. Figure 6.5 depicts the improvement of node start times achieved by the *Snapshot Start*.

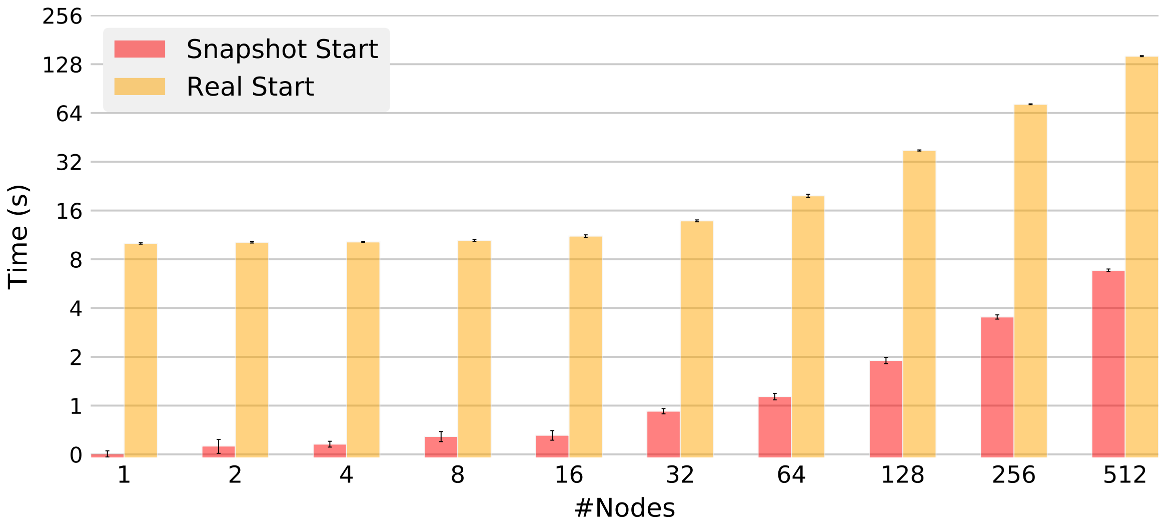


Figure 6.5: Boot Modes Debian

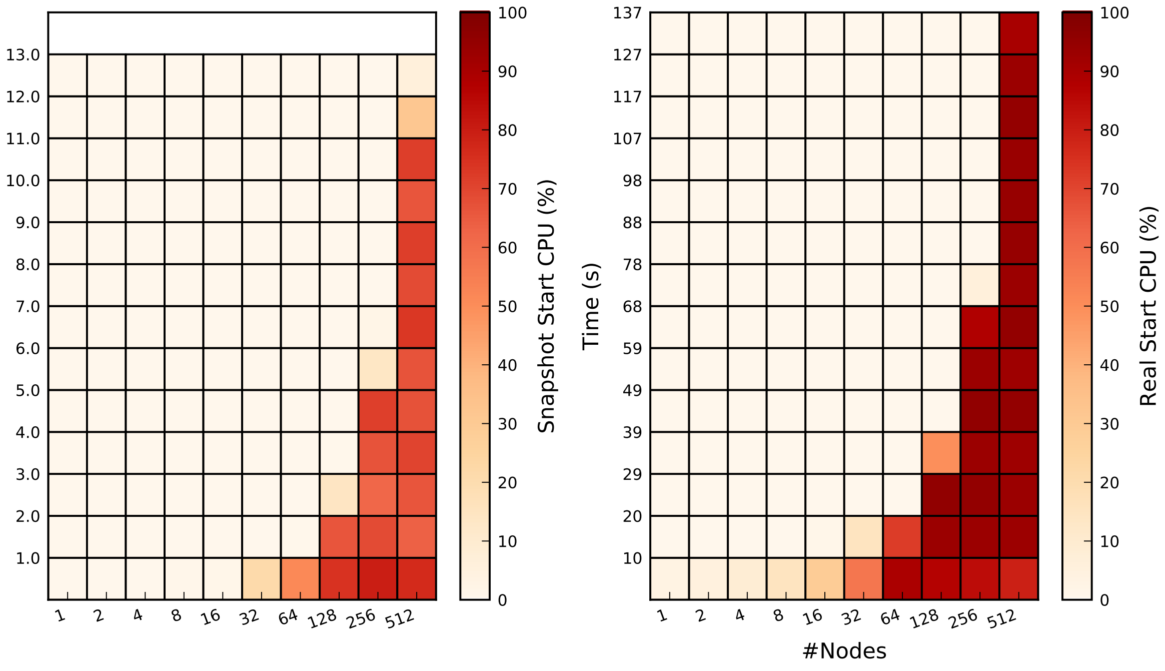


Figure 6.6: Boot Modes Debian CPU

Even for 512 VMs, only a few seconds are required to restore the state of the VM (6.8s). A full boot requires 143.3s, hence the node start times are reduced by factor 21. Additionally, the CPU requirements are lower as outlined by Figure 6.6.

6.3 QEMU NIC Models Bandwidth

CPU is not the only critical resource while using MiniWorld. The network bandwidth of the *Qemu NIC Models* varies. The following study tries to figure out which *Qemu NIC Models* offers the best network bandwidth. The supported *Qemu NIC Models* can be displayed with

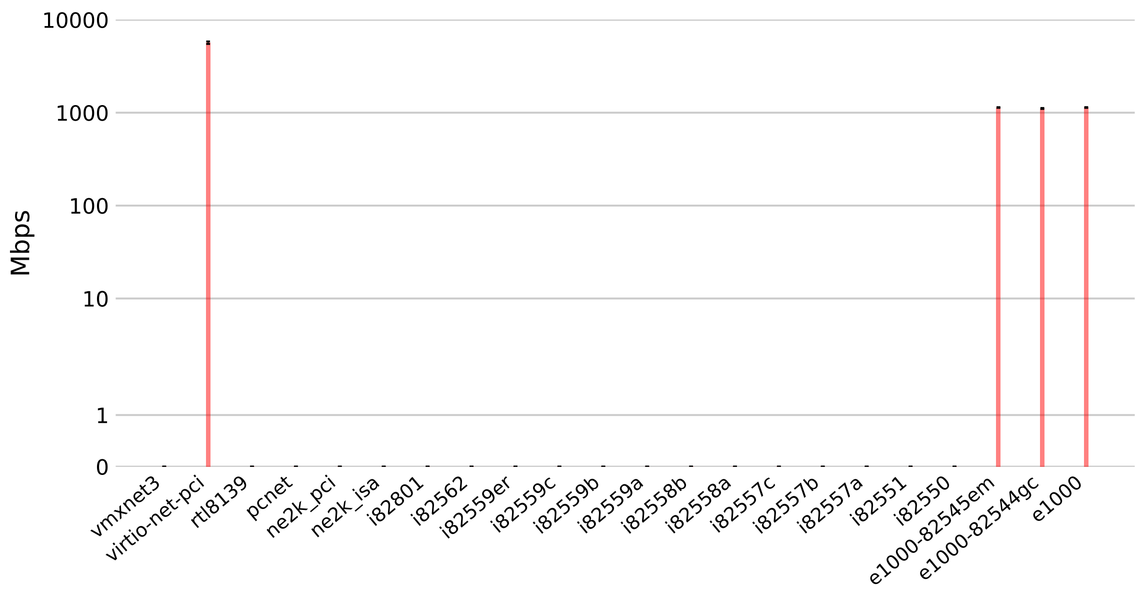


Figure 6.7: Qemu NIC Models Bandwidth

the command shown in Listing 33.

```
kvm -device \? 2>&1 | awk '/Network devices/,/Input devices/' | grep name |
↪  awk '{print $2}' | tr -d '",'
```

Listing 33: Get Qemu NIC Models

For each model, two VMs have been started. The model can be supplied to the *Scenario Config* with the key qemu→nic→model. The *iperf* command is started on node 1 via the *Pre Network Shell Commands*. From node 2, the command *iperf -f mbits -t 60 -c 10.0.0.1* is executed and displays the bandwidth tested with a Transmission Control Protocol (TCP) test in 60s. The network backends *rocker*, *usb-bt-dongle*, *usb-net*, *virtio-net-device*, *virtio-bus* may need further setup or are not working at all. Starting with these models resulted in a error by Qemu. Therefore, they are not part of the study.
Figure 6.7 depicts the bandwidth for each *Qemu NIC Model*. Some of the drivers seem to be even not available or working with *OpenWRT Barrier Braker*. Errors during boot such as *SIOCSIFADDR: No such device* and *SIOCSIFFLAGS: Cannot allocate memory"* result in 0 bandwidth in the Figure. The derivates of the *e1000* driver show a good performance greater than 1000 Mbps but the fastest driver is the paravirtualized *virtio-net-pci* driver. Therefore, it should be the preferred *Qemu NIC Model*.

6.4 Network Backends

The 4 presented network backends are evaluated in terms of bandwidth and Round Trip Times (RTTs) in the following. Afterwards, a showcase for the *WiFi* network backend is

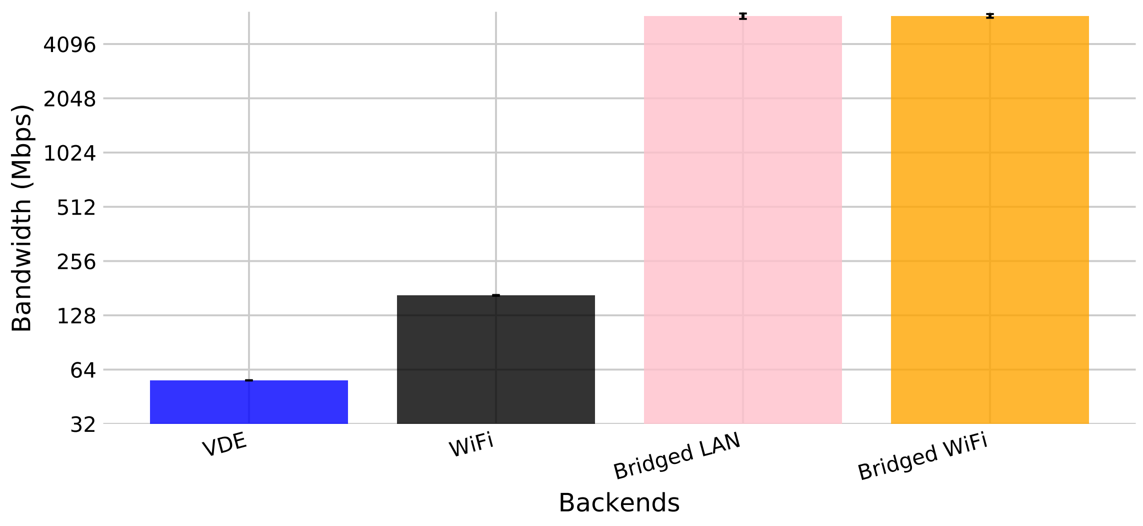


Figure 6.8: Network Backend Throughput

depicted.

6.4.1 Bandwidth

For the study of bandwidth, 2 VMs are used. On node 1 *iperf* is started in server mode. Node 2 connects via TCP and takes 60 seconds to determine the available bandwidht. All nodes have been started with 1024MiB RAM and the *virtio-net-pci Qemu NIC Model*. Moreover, each node is given only one CPU core.
The study is depicted in Figure 6.8. The **VDE** network backend provides only a bandwidth of 55.7 Mbps on average. Both *vde_switch* processes required 100% on a CPU core. Further researched pointed out that the bandwidth correlates with the number of ports, at least in the hub mode. No tests for the switch mode has been done. Booting two Qemu nodes manually and interconnecting them with VDE with the default port size (32), resulted in a total bandwidth of 439 Mbps. It is not sure whether this is intended behaviour since the hub seems to flood traffic even for non active ports. Maybe this is the result of the color patch. Further research is required to improve the integration of VDE into MiniWorld.
The **WiFi** network backend offers an average bandwidth of 165.1 Mbps. This is enough to simulate 802.11/a/b/g but is not sufficient for 802.11n/ac networks. The limiting factor is the CPU. Providing the VMs with more cores may increase bandwidth also. The multicast transport mimics the wireless broadcast nature but also poses high CPU requirements since every node receives the traffic. Since there is no link quality impairment yet, the situation is even worse. Note that there is also a TCP transport builtin which could be used to redirect traffic to the host. A user-space application could when apply link impairment. Further research is needed to improve performance and offer link emulation capabilities. Nevertheless, it is very useful to simulate 802.11 specific software. A use case of the *WiFi* network backend is demonstrated in Section 6.4.3.
The *Bridged* network backends differ only slightly: The **Bridged LAN** provides a bandwidth of 5848.5 Mbps while the **Bridged WiFi** network backend offers 5867.6 Mbps of bandwidth. This is interesting since *ebtables* does not seem to reduce bandwidth. But since the standard derivation is 204.1 and 143 respectively, this may the due to the measurement even though

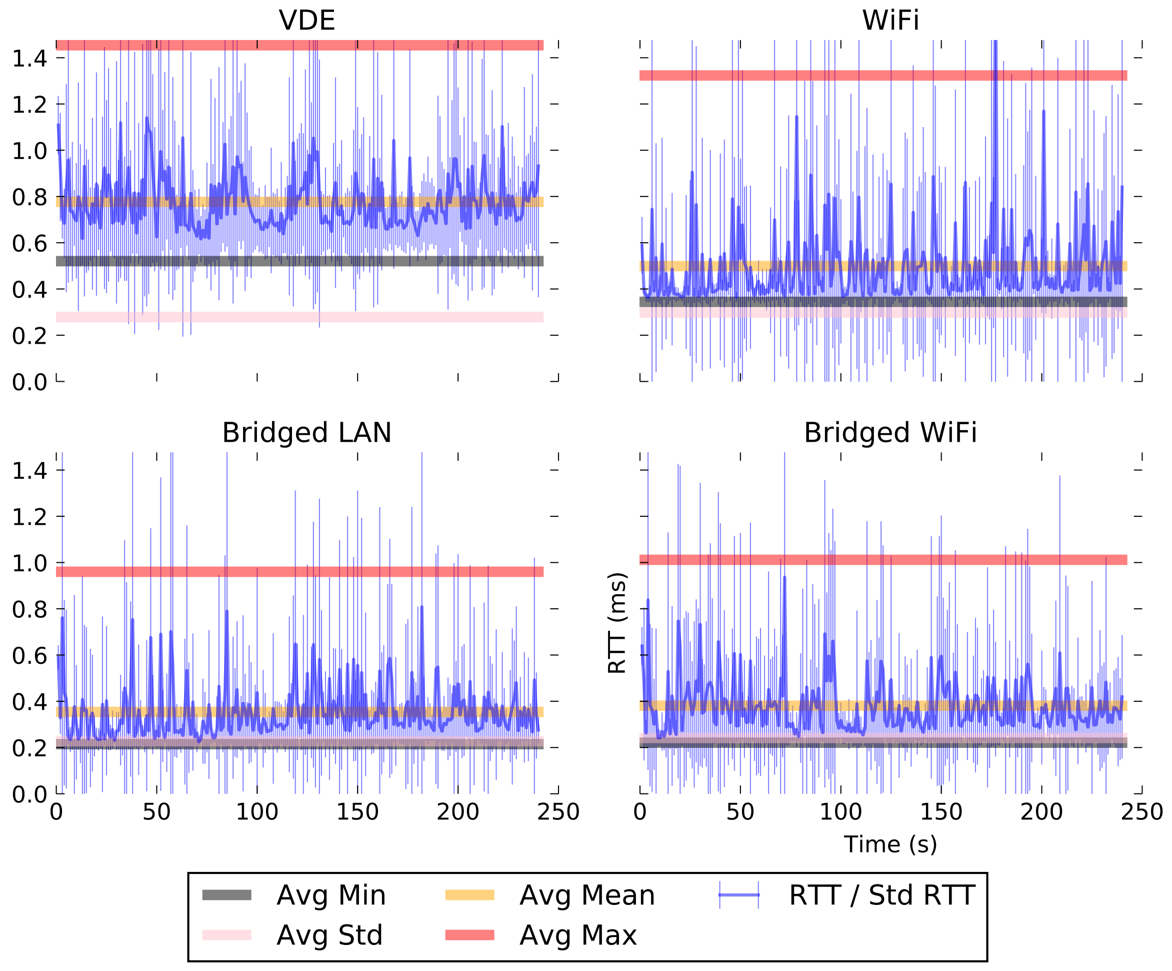


Figure 6.9: Network Backend Delays

it has been repeated 10 times and the iperf measurement has been given 240 seconds for the TCP bandwidth test.

6.4.2 Round Trip Times

Not only the bandwidth of network backends is important. The delay of packets is crucial to some protocols. Therefore, the RTT of all networks backends has been examined. Figure 6.9 shows the results gather by the *ping* command over 240 seconds. All 4 subplots share the same x and y axis. The red line depicts the average maximum delays since the study has been conducted 10 times. The *avg* is therefore taken from these runs. With slightly more than 1.4 ms the *Avg Max* (red line) is the highest for *VDE*. The *Bridged* network backends provide the same delay characteristics since both use Linux bridges. The *Avg Mean* (yellow line) of both is less than 0.4 ms. The *WiFi* network backend has approximately 0.5ms as the *Avg Mean* value. The highest *Avg Mean* value has the *VDE* network backend (approximately 0.8ms). Since the number of ports is very high for the *VDESwitches* by default (65537), this might have influenced the measurement in the same way it has for the bandwidth experiment. In summary, all network backends provide good RTTs values.

6.4.3 Distributed mac80211_hwsim showcase

The *WiFi* network backend is the only network backend which provides real WiFi NICs to guests[6]. Most routing protocols do not require WiFi interfaces. Even though B.A.T.M.A.N. operated at layer 2, a normal lan device is sufficient since no characteristics of the wireless channel are required. This is different for *802.11s* which has a modified MAC layer.
To present the feasibility of the *WiFi* network backend, a study has been conducted with *802.11s*. A *Chain 32* topology has been built by controlling with whom a WiFi station is allowed to peer itself. The command[7] in Listing 34 can be used to control the routing.

```
iw dev <device> station set <mac> plink_action block
```

Listing 34: 802.11s Peer Link Blocking

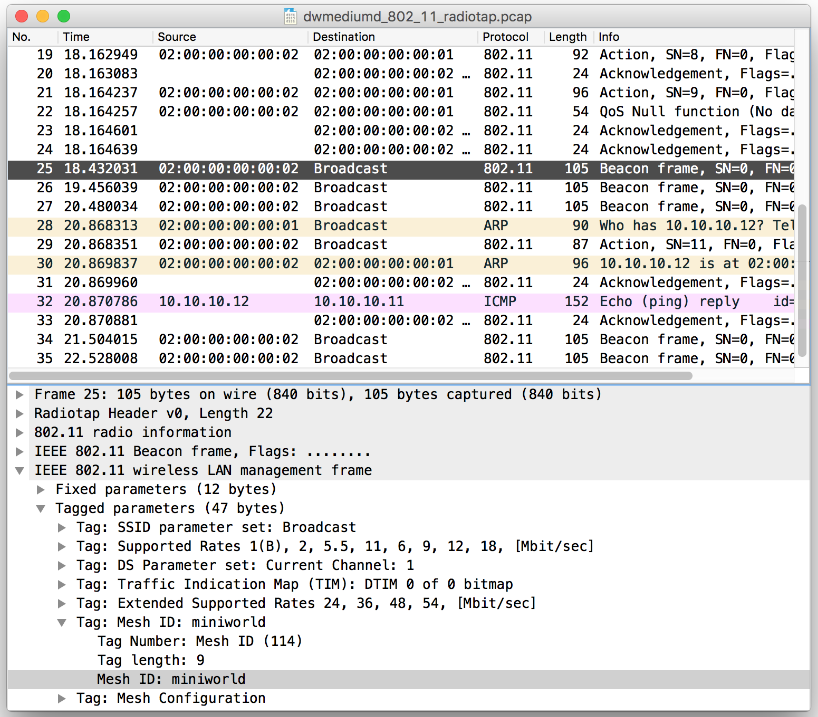
For all except the direct neighbours, link peering is disabled, hence a *Chain* topology is created.
Listing 35 shows the *Pre Network Shell Commands* used for the study.

```
{
  "provisioning":
    "shell": {
      "pre_network_start": {
        "shell_cmds" : [
          // configure 802.11s
          "modprobe mac80211_hwsim radios=1",
          "ip link set wlan0 down",
          "ifconfig wlan0 hw ether 02:00:00:00:00:‘printf ’%02x’
 ↪  {node_id}‘",
          "iw dev wlan0 set type mp",
          "ip link set wlan0 up",
          "iw dev wlan0 mesh join miniworld freq 2412",
          // wifi ip
          "ip addr add 10.10.10.{node_id}/24 dev wlan0",

          // multicast routing
          "ip addr add 172.21.0.{node_id}/16 dev eth0",
          "ip l s dev eth0 up",
          "route add -net 224.0.0.0 netmask 224.0.0.0 dev eth0",

          // start wmediumd
          "cd /root/wmediumd/ && wmediumd/wmediumd -c 32.cfg &"
        ]
      }
```

[6]Limited to Linux since the network backend requires the mac80211_hwsim kernel module.
[7]`https://github.com/o11s/open80211s/wiki/HOWTO`. Last viewed on 02.12.2016.



```
dwmediumd_802_11_radiotap.pcap
No.  Time       Source             Destination          Protocol  Length  Info
 19  18.162949  02:00:00:00:00:02  02:00:00:00:00:01    802.11     92  Action, SN=8, FN=0, Flag
 20  18.163083                     02:00:00:00:00:02 …  802.11     24  Acknowledgement, Flags=.
 21  18.164237  02:00:00:00:00:02  02:00:00:00:00:01    802.11     96  Action, SN=9, FN=0, Flag
 22  18.164257  02:00:00:00:00:02  02:00:00:00:00:01    802.11     54  QoS Null function (No da
 23  18.164601                     02:00:00:00:00:02 …  802.11     24  Acknowledgement, Flags=.
 24  18.164639                     02:00:00:00:00:02 …  802.11     24  Acknowledgement, Flags=.
 25  18.432031  02:00:00:00:00:02  Broadcast            802.11    105  Beacon frame, SN=0, FN=0
 26  19.456039  02:00:00:00:00:02  Broadcast            802.11    105  Beacon frame, SN=0, FN=0
 27  20.480034  02:00:00:00:00:02  Broadcast            802.11    105  Beacon frame, SN=0, FN=0
 28  20.868313  02:00:00:00:00:01  Broadcast            ARP        90  Who has 10.10.10.12? Tel
 29  20.868351  02:00:00:00:00:02  Broadcast            802.11     87  Action, SN=11, FN=0, Fla
 30  20.869837  02:00:00:00:00:02  02:00:00:00:00:01    ARP        96  10.10.10.12 is at 02:00:
 31  20.869960                     02:00:00:00:00:02 …  802.11     24  Acknowledgement, Flags=.
 32  20.870786  10.10.10.12        10.10.10.11          ICMP      152  Echo (ping) reply    id=
 33  20.870881                     02:00:00:00:00:02 …  802.11     24  Acknowledgement, Flags=.
 34  21.504015  02:00:00:00:00:02  Broadcast            802.11    105  Beacon frame, SN=0, FN=0
 35  22.528008  02:00:00:00:00:02  Broadcast            802.11    105  Beacon frame, SN=0, FN=0

▶ Frame 25: 105 bytes on wire (840 bits), 105 bytes captured (840 bits)
▶ Radiotap Header v0, Length 22
▶ 802.11 radio information
▶ IEEE 802.11 Beacon frame, Flags: ........
▼ IEEE 802.11 wireless LAN management frame
  ▶ Fixed parameters (12 bytes)
  ▼ Tagged parameters (47 bytes)
    ▶ Tag: SSID parameter set: Broadcast
    ▶ Tag: Supported Rates 1(B), 2, 5.5, 11, 6, 9, 12, 18, [Mbit/sec]
    ▶ Tag: DS Parameter set: Current Channel: 1
    ▶ Tag: Traffic Indication Map (TIM): DTIM 0 of 0 bitmap
    ▶ Tag: Extended Supported Rates 24, 36, 48, 54, [Mbit/sec]
    ▼ Tag: Mesh ID: miniworld
        Tag Number: Mesh ID (114)
        Tag length: 9
        Mesh ID: miniworld
    ▶ Tag: Mesh Configuration
```

Figure 6.10: Wireshark Capture Of 802.11s Beacon Frames and ICMP Ping

```
25      }
26  }
```

Listing 35: WiFi Network Backend 802.11s Scenario Config

First the *mac80211_hwsim* is loaded (line 7) with 1 radio device. The MAC address has to be changed (line 9) since they are the same for each node (02:00:00:00:<nr>:00). The {*node_id*} string is replaced by MiniWorld with the actual node id (starting with 1). The mesh mode is activated in line 10. A mesh network with the name *miniworld* is created in line 12. IP addresses are assigned from the 10.10.10.0/24 subnet. Multicast routing (lines 16-19) is required so that 802.11s frames can be distributed to all nodes via a multicast group. Finally, the modified *wmediumd* binary is started (line 22). The config file *32.cfg* contains the MAC mapping between node ids and the MAC addresses and is required by *wmediumd*. In future releases it may get deleted.
Figure 6.10 shows a *Wireshark* capture on the *hwsim0* device which is in monitor mode such that all frames can be received. A *radiotap* header is added to the frames so that WiFi specific settings such as bandwidth and signal strengths are viewable. The *802.11* protocol can be seen in the *Protocol* row. Moreover, Figure 6.10 highlights the id of the mesh network in the detail view of the bottom of the Figure.

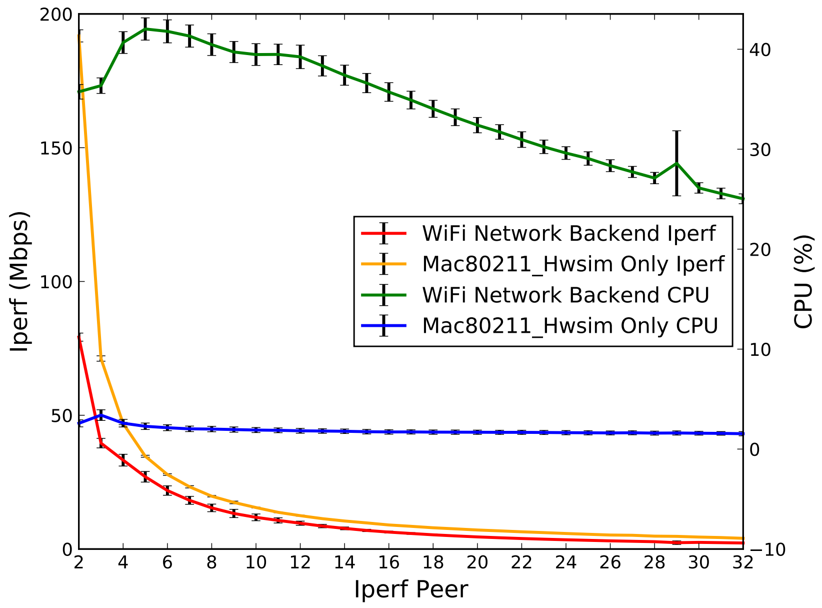


Figure 6.11: Distributed mac80211_hwsim Showcase

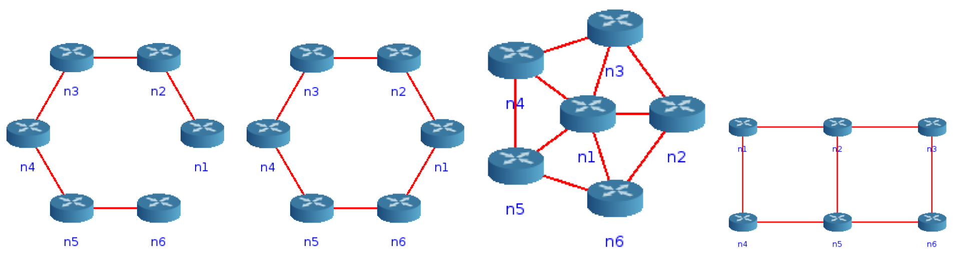
```
root@miniworld32:~# iw dev wlan0 mpath dump
DEST ADDR         NEXT HOP          IFACE       SN      METRIC   QLEN
 ↪  EXPTIME          DTIM     DRET     FLAGS
02:00:00:00:00:01 02:00:00:00:00:1f wlan0       5       245961   0
 ↪  3476     0          0x15
02:00:00:00:00:1f 02:00:00:00:00:1f wlan0       0       8193     0
 ↪  3476     0          0x11
```

Listing 36: 802.11s Mpath Dump

After the *Chain 32* topology has been started, routing has been verified by pinging nodes 1 and 32 from node 32. Listing 36 outlines that the route to 02:00:00:00:00:1f (node 31) is through node 31. The same applies for node 1 (02:00:00:00:00:01). Moreover, the *Metric* is depicted in the Listing and is worse for the path to route 1 compared to the route to node 31.

To prove that the modifications to *wmediumd* did not change crucial aspects of the wireless NIC, its behaviour has been compared with *mac80211_hwsim*. For that purpose, the same *Chain 32* topology is set up by creating 32 LXC namespaces and putting each WiFi device into the appropriate namespace. For that, a test script[8] from *wmediumd* has been modified. Figure 6.11 outlines the results of performing an iperf measurement from node x to node 1. Traffic is therefore routed along the chain. The measurement is based on TCP with a duration of 240 seconds. The results are identical for the *WiFi* network backend study (red line) and the mac80211_ hwsim experiment (yellow line). The bandwidth decreases while the size of the routing chain increases. Interestingly the bandwidth is more than 2 times higher for *mac80211_hwsim* for x=2. Note that the VMs are only given 1 CPU core while the

[8] `https://github.com/bcopeland/wmediumd/blob/master/tests/n-linear-mesh.sh`



(a) Chain 6 (b) Cycle 6 (c) Wheel 6 (d) Grid 6

Figure 6.12: Differential Switching

LXC namespaces are not limited in terms of CPU. The *WiFi* network backend creates higher CPU load (green line) due to *full system virtualization* and the modified *wmediumd* binary. Interesting is that even the container version did not result in higher throughput since CPU is not the bottleneck.

6.4.4 Connection Switching Times

Mobile nodes change their positions frequently, hence switching between different topologies needs to be fast. The following Section investigates the network switching of the *Bridged* network backends since the *WiFi* network backend does not offer connection switching at the time of writing and the *VDE* network backend did not show good enough results such that it is usable as a network backend for wireless emulation. Additionally, the *VDESwitch* seems to have a hiccup if the process is running with 100% CPU utilization on one core. The result is that the *REPLable* mechanism had to send commands several times until they have been processed.
Note that in the following experiments the management network has not been taken into account because it is only created once after all nodes are started. Moreover, its setup relies on the same commands and techniques which are evaluated in the following.
Different topologies have been created via the Core UI with the help of the topology generator. Figure 6.12 depicts the topologies with 6 nodes: In 6.12a a *Chain 6* is built. A *Cycle 6* topology (6.12b) includes additionally a link between the first and the last node. By connecting each node with the first one, a *Wheel 6* has been created (6.12c). A *Grid 6* topology (6.12d) cannot be simply changed from a *Wheel 6* topology. The number of 6 nodes were only chosen for the graphical representation. Figure 6.13 outline the results from switching between these topologies but with 128 nodes instead. The *Execution Mode* is *iproute2* with all *Execution Options* enabled. Moreover, the *CORE Mobility Pattern* is used to switch between *CORE* XML topologies.
The first two bars of each bar chart show the **Differential Network Switching** capability, since only the differences between topologies are changed. For example switching between *Chain 128* and *Cycle 128* requires less than a second (red bar at *Cycle 128*) since the *Step Time* of the *RunLoop* is 1 second by default. Therefore each step takes at least one second. The *Differential Network Switching* feature is provided by the *NetworkManager* and hence every network backend benefits from it automatically.
The values of the green and blue lines have been created by defining the appropriate topology as the only one in the *Scenario Config*. Therefore no *Differential Network Switching* can be performed. In all topology switching cases depicted in 6.13, *Differential Network Switching* is faster compared to the *Full Network Switching*. The used network backend is the *Bridged WiFi*

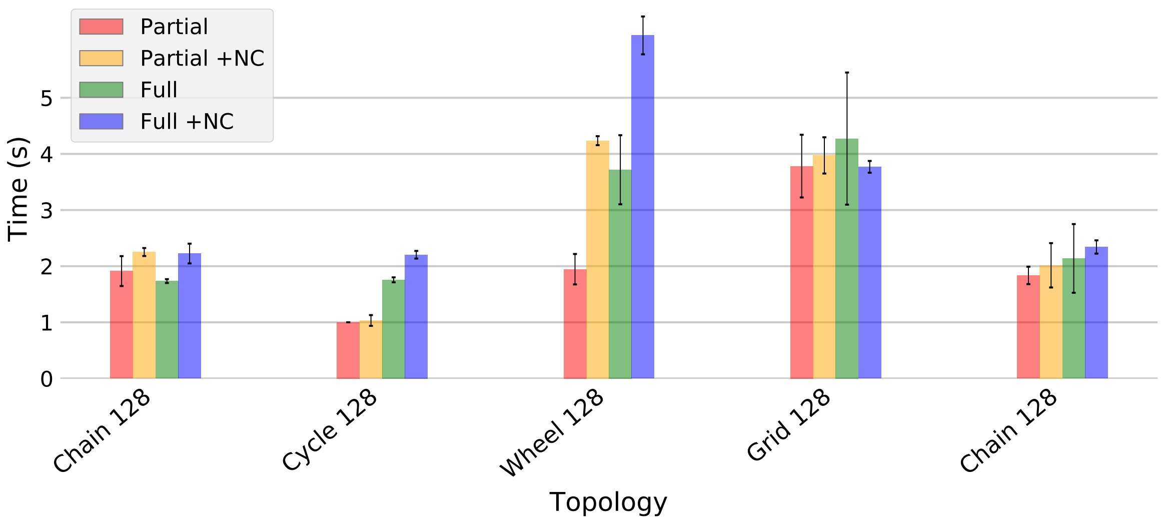


Figure 6.13: Network Switching Topologies (WiFi Network Backend)

network backend. Moreover, the Figure illustrates that Network Checking (NC) increases switching times since NICs need to configured in terms of IP addresses and the network connectivity needs to be checked with the *ping* command. Since **Network Checking** is an important feature to ensure that a network topology has been switched correctly, the additional times can be neglected.
The connection switching times of the *Bridged* network backends and especially the **Execution Modes** and **Execution Options** are further investigated in the following.

6.4.5 Bridged WiFi Network Backend

The connection switching times of the *Bridged WiFi* network backend are depicted in Figure 6.14. Note that for the *Pyroute* execution mode, the transactional database *IPDB* worked for smaller topologies but showed errors for the *Grid 128* topology used by the study depicted in 6.14. The problem[9] has been reported and is discussed on GitHub at the time of writing. On the x-axis the *Execution Modes* are shown whereas the *Execution Options* and combinations of them are illustrated by each bar color. The abbreviations for the *Execution Options* are as follows: *P* stands for *Parallel*, *B* for *Batch* and *O* for *One Shell Call*.
For all *Execution Modes* the sequential mode is the slowest (-/-/-). Adding the *One Shell Call* option (-/-/O) reduces the amount of time by approximately factor 4 and more for all *Execution Modes*. The *One Shell Call* option reduces overall switching times since for each second bar the times are reduced.
Note that for the **Pyroute (IPBatch)** execution option the *Execution Options* only effect the *ebtable* command which is used to govern over connectivity. The batch mode of Pyroute2 is enabled by default. Therefore, the influence of the *Execution Options* to **ebtables** are depicted in the *Pyroute2 (IPBatch)* execution option. The *One Shell Call* option reduces the number of *ebtable* commands (-/-/O). The batch mode allows to communicate only once with the kernel to update the tables and adds another performance improvement (-/B/O). Running the *ebtable* commands in parallel (P/-/-) improves performance compared to (-/-/-) only a little. The reason for this might be the lock which is used by the *atomic file* to store *ebtable*

[9] `https://github.com/svinota/pyroute2/issues/304`

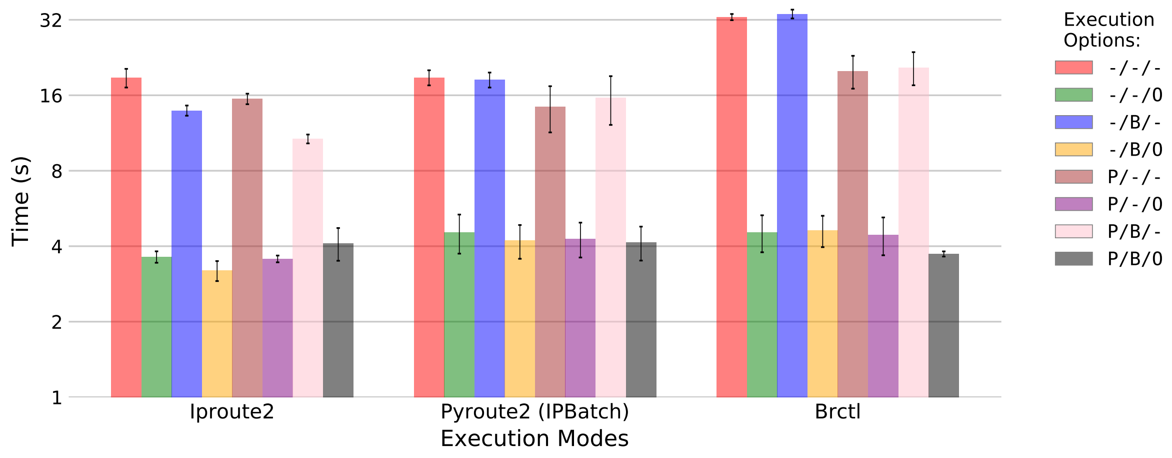


Figure 6.14: Network Switching Modes Single

table updates. Note that MiniWorld enables the locking mechanism by default. A more improved batch mode for *ebtables* where switching commands are only sent once to the program (for example via stdin) could further improve switching times.
The **Brctl** command has no batch mode, hence (-/B/-) does not improve performance for the *Brctl* execution mode. The parallel mode improves performance a little (P/-/-) compared to (-/-/-), but the *One Shell Call* execution option shows again the best performance improvements (-/-/O).
The same arguments apply for the **Iproute2** execution mode. Since *Iproute* has a batch mode the combination (-/B/O) further improves performance.
For all except *Iproute2* the (P/B/O) combination is the fastest. For Iproute2 the combination (-/B/O) is the fastest, but since the *One Shell Call* execution mode is prioritized over the *Parallel* execution mode this may be due to outliers since the standard derivation is is high for all measured times.

6.4.6 Bridged LAN Network Backend

Figure 6.15 depicts the switching times for the *Bridged LAN* network backend. The sequential mode (-/-/-) is very slow since for every connection a bridge and two NICs are required. The batch mode of **Iproute2** (B/-/-) enables commands to be run in one batch operation, at least for the *Iproute2* execution mode. Note that *ebtables* is not required for the *Bridged LAN Network Backend*, hence the batch mode of *Iproute* can be measured alone. The *One Shell Call* (-/-/O) and *Parallel* (P/-/-) execution options do not achieve as good switching times as (B/-/-) does. Since the batch mode is prioritized over the other options, the fastest switching times are all combinations where the batch mode is involved.
In the **Brctl** execution mode, all bridge related commands require a *brctl* call. Therefore, only changing the NIC state can be performed by *Iproute* and thus also its batch mode. Again, the *One Shell Call* execution mode shows better performance compared to the *Parallel* execution mode even though there is no lock for the *brctl* command as there is for the *ebtables* command.
The switching times for the **Pyroute2 (IPBatch)** execution mode are the same for all combinations since they do not affect it and the batch mode is enabled by default.

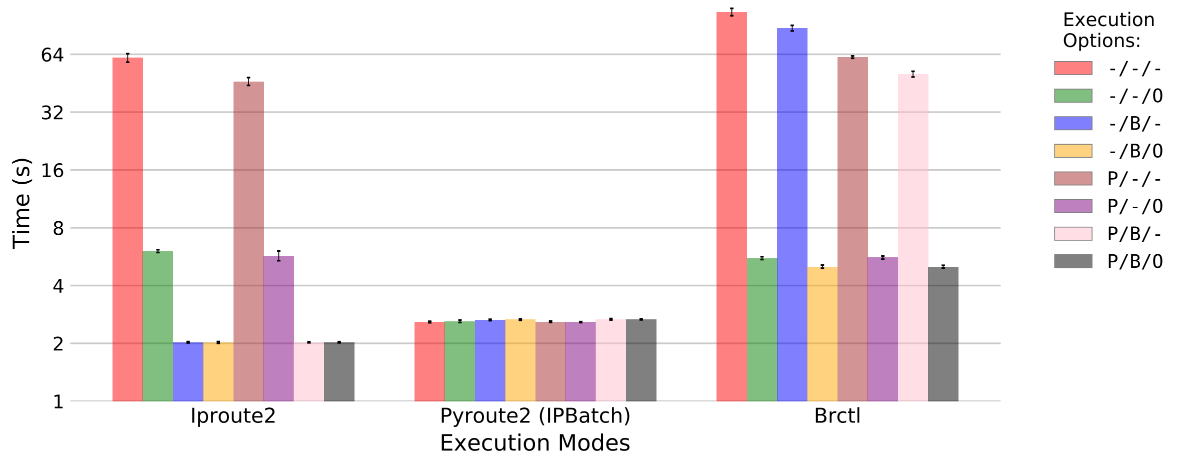


Figure 6.15: Network Switching Modes Multi

6.4.7 Summary

To sum it up, one should always enable all *Execution Modes*. All commands which support a batch mode make use of it. *One Shell Call* is prioritized over the *Parallel Execution Option* since both parallelize command execution but *One Shell Call* showed better performance. Since the *Parallel* execution option is slower than the *One Shell Call Option*, there is no reason to keep it inside MiniWorld.
Iproute2 emerged as the fasted switching *Execution Mode* but requires a correct version to be compiled. *Brctl* worked always correct without any modification or updates required. The *Pyroute2* execution mode may be a good compromise between both since it offers good switching times and did not cause any problems in the tested version. Moreover, there is less complexity in the implementation.
The switching for the *Bridged LAN* network backend is faster compared to the *Bridged WiFi* network backend since *ebtables* is not involved. The experiments did not include the Network Checking for the network backends. For new connections in the *Bridged LAN* network backend the NICs in the VMs need to be configured. This could be changed since the connections are known beforehand, but removes the dynamic character of the network backend. In contrast, the NICs in the *Bridged WiFi* network backend can be configured once in the first step.

6.5 Distributed Emulation

In the following the distributed mode of MiniWorld is evaluated. Initially, the coordination from the *Coordinator* with its *Clients* is examined in Section 6.5.1. For further tests, a new test system is introduced in Section 6.5.2. The boot times of nodes which are distributed across a cluster is studied in Section 6.5.3. The network switching times are examined in Section 6.5.4. Since tunnels between clients may introduce further delays, they are studied in Section 6.5.5.
Finally, a study with the *Serval*[10] Delay-Tolerant Networking (DTN) software is presented in Section 6.5.6. 7 computers in total allowed to setup a *Chain 256* network topology. All distributed experiments have been performed with Docker containers. For that purpose,

[10]`http://www.servalproject.org/`

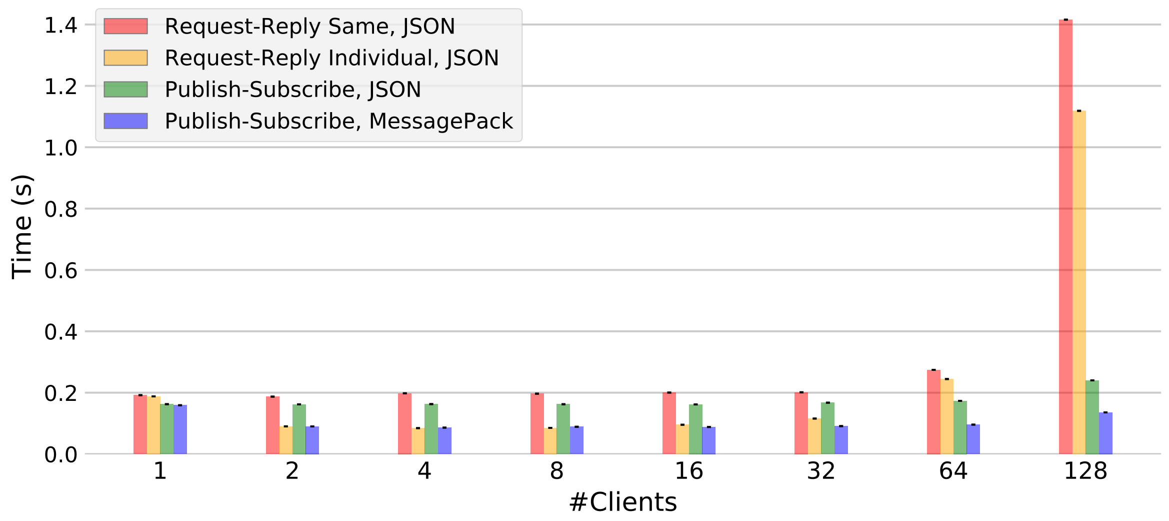


Figure 6.16: Distributed Coordination Step Times

on each client a MiniWorld *Docker* container has been deployed. To allow communication among the containers on the IP layer (for the *Gretap* tunnels), the *–net=host* option from Docker has been used so that containers are not put into an extra network namespace.
If not stated explicitely, the *Publish-Subscribe* communication pattern with *MessagePack* serialization is used. Moreover, *iproute2* with all *Execution Modes* and *Execution Options* is used.

6.5.1 Distributed Coordination

The bottleneck in the distributed mode is the communication of the coordinator with its *clients* (server hosting a MiniWorld instance which is running in client mode) as well as the performance of tunnels between *virtual nodes* (node in MiniWorld). The step times across the network have been evaluated with Docker containers. This enabled the simulation of arbitrary number of servers but does not incorporate real network conditions. Since nodes are started in parallel, not more than 128 servers could be simulated because then each server starts one virtual node. Therefore 2 *virtual nodes* per core are started in parallel. Higher number of *clients* pushed the test system to its limits.
Figure 6.16 points out the time required to coordinate the clients. The distance matrix is sent by the *coordinator* either via the *Request-Reply* or the *Publish-Subscribe* pattern. After a *client* has performed all its actions associated with a *step* (node starting, connection switching, etc.), it syncs with the *coordinator* so that all *clients* are in sync before a new *step* is performed by the *coordinator*. These step times are depicted in Figure 6.16. To visualize step times smaller than 0, the *step time* has been set to 0. Therefore, the *coordinator* waits 0 seconds after is has synced all *clients* before it performs a new *step* call. The **Request-Reply** pattern (red and yellow lines) is slower for all number of clients. Sending each *client* the same distance matrix (red line) is faster than filtering out the relevant parts for the *clients* at the *coordinator* side (yellow line). The reason for this is that the distance matrix of the *Chain 128* topology is not very big. Filtering out the relevant parts of the distance matrix at *client* side parallelizes the filtering. The performance gains of the **Publish-Subscribe** pattern shows especially for 128 *clients* where the *Request-Reply* pattern takes over a second (more than a default time step in MiniWorld) whereas the *Publish-Subscribe* pattern requires less than

Resource	Value
Host OS	Ubuntu 16.04.1 LTS
Docker OS	Ubuntu 14.04.5 LTS
Virtualization Technology	KVM
Kernel	4.4.0-38-generic
RAM	2x DIMM DDR3 Synchronous 1600 MHz 8GiB
CPU	Intel(R) Core(TM) i7-4771 CPU @ 3.50GHz
CPU Cores Physical/Virtual	4/8
Network	Intel Corporation Ethernet Connection I217-LM (rev 04)
Iproute2	Git release v4.2.0
Qemu	Git release v2.7.0

Table 6.3: Technical Data Test System 2

a quarter second. Additionally, serializing messages with *MessagePack* instead of *JSON* reduces further coordination time (blue line).

6.5.2 Test System 2

The following experiments are based on another test system. The distributed mode of MiniWorld is evaluated with 5 computers which are virtualized by KVM. The technical details of the computers are homogenous and are depicted in Table 6.3. Each computer has a Core i7 processor with 4 physical and 8 virtual cores. Hence, 8 Qemu processes are started in parallel. Moreover, it has 16 GiB of memory and a gigabit ethernet card. In contrast to *Test System 1*, a newer Ubuntu LTS release is used. Therefore, also a newer kernel is used.

6.5.3 Boot Times

The following experiment demonstrates the CPU bottleneck which occurs if too much VMs are run on a single machine. *Test System 2* has been used for the experiment. 5 KVM VMs are used in MiniWorld's distributed mode as clients. For that purpose, MiniWorld is deployed by means of Docker on each client. 40 OpenWrt Barrier Braker nodes are started by n clients where n is increased by 1 until all 5 clients take part in the distributed emulation.
The boot times (*Selectors Boot Prompt*) are depicted in Figure 6.17. Nested virtualization poses high CPU resources since 40 nodes could be started in 58.4s on the same machine type but bare-metal. The start of 40 nodes with a single KVM virtualized client took 640.2 seconds (red bar). The first VMs are started faster than the last ones since the already started VMs show high CPU usage even in idle mode.
Doubling the number of clients reduces the boot times to 263.7 seconds (factor 2.4). With a total of 5 clients, the boot times could be lowered to 95.8 seconds. Hence, the total boot times could be reduced by factor 6.7. The savings are huge since the clients resources are overcommitted with 40 nodes.
Even for only 20 nodes, the boot times could be reduced by factor 5.4 (from 259.3 s to 47.8s). The *Snapshot* boot mode could not be evaluated at all, since KVM showed a bug[11]. The bug

[10]`https://github.com/svinota/pyroute2`. Commit ID: commit 13f1adb1ab2d5ca8927f1eb618bbb9317-0b61c33
[11]`https://bugs.launchpad.net/ubuntu/+source/linux/+bug/1530405`. Last viewed on 12.12.2016.

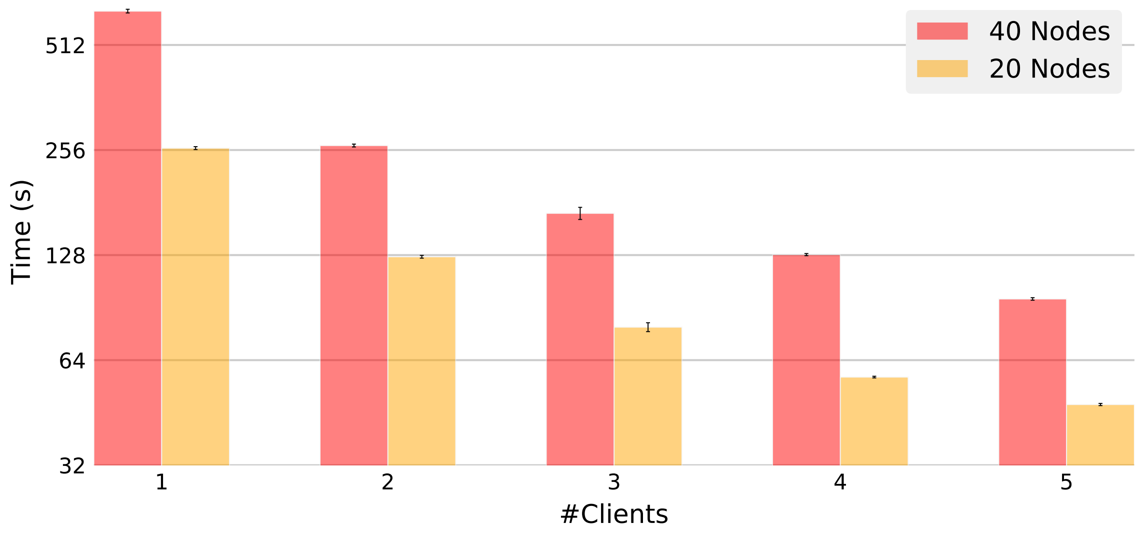


Figure 6.17: Distributed Boot Times, 40 OpenWrt BB Nodes, RamDisk, 128MB RAM, Selectors Boot Prompt, NodeDistributionEqual, Test System 2

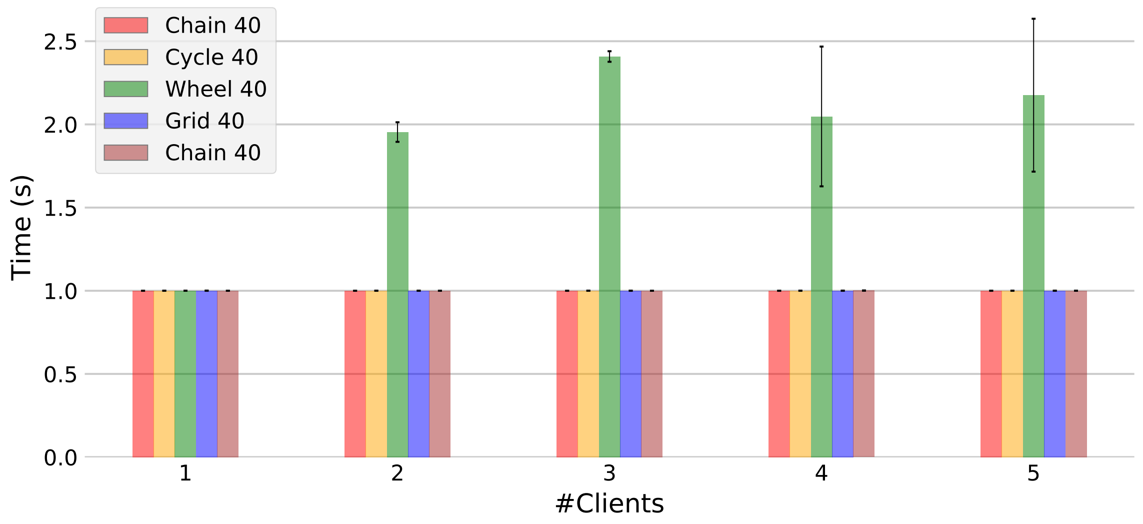


Figure 6.18: Distributed Differential Network Switching, 40 OpenWRT BB Nodes, Test System 2, Fixed-Range Model, Bridged WiFi Network Backend, No Link Quality Impairment, 2 Runs Only

did not arise for *Test System 1*, hence it may be due to a different kernel version. Even Qemu version 2.7.0 did not solve the problem.
Even though 5 clients have been used in the distributed mode, an unvirtualized computer is still faster at booting 40 OpenWrt Barrier Braker nodes (58.4s). Hence, nested virtualization should be avoided.

6.5.4 Network Switching

The network switching times of the distributed mode are depicted in Figure 6.18. 5 clients have been used to run 40 OpenWrt Barrier Braker nodes with the *Bridged WiFi* network backend and the *Fixed-Range* model without any *Link Impairments*. The figure points out that without any tunnels, the time of a time step (1 second), is not exceeded. Only the *Wheel 40* topology requires more than 1 second since for every connection to node 1 (39), a tunnel has to be created.

6.5.5 Tunnel Delays

Tunnels introduce overhead, since frames need to be encapsulated by a PDU.

```
ip route add 10.0.0.3 via 10.0.0.2
echo 1 > /proc/sys/net/ipv4/ip_forward
```

Listing 37: Setup Routing For Node 1 To Node 3 Via 2

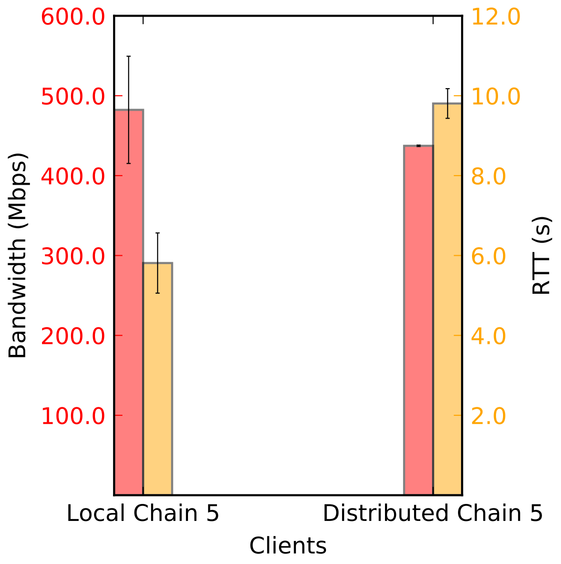


Figure 6.19: Distributed Mode: Tunnel Overhead, 5 OpenWRT BB Nodes, Test System 2, Fixed-Range Model, Bridged WiFi Network Backend, No Link Quality Impairment

For *Gretap*, each frames is wrapped into an IP packet. The following experiment examines the overhead which tunnels pose on bandwidth and delay. A *Chain 5* topology is used which routing setup between all nodes. Routing has been configured with the commands depicted in Listing 37. Additionally, all OpenWrt configured iptables rules are removed after VM boot. 5 clients and 5 OpenWrt Barrier Braker nodes are used. The x-axis depicts the number of clients. The scheduling algorithm is *NodeDistributionEqual*. Hence, the nodes are equally scheduled on the *clients*. Since not the same number of nodes can be placed on all *clients*, the remaining ones are scheduled round-robin. Delay is studied with the *ping* command whereas the bandwidth is measured with *iperf*. Both tests connected from node 1 to node 5 and recorded data in a 60s experiment which has been repeated 10 times.
For x=1, all nodes are placed on one client only. Hence no tunnels are used at all. The examined average bandwidth is 482.3 Mbps. The average delay is 5.8ms. If the *Chain 5*

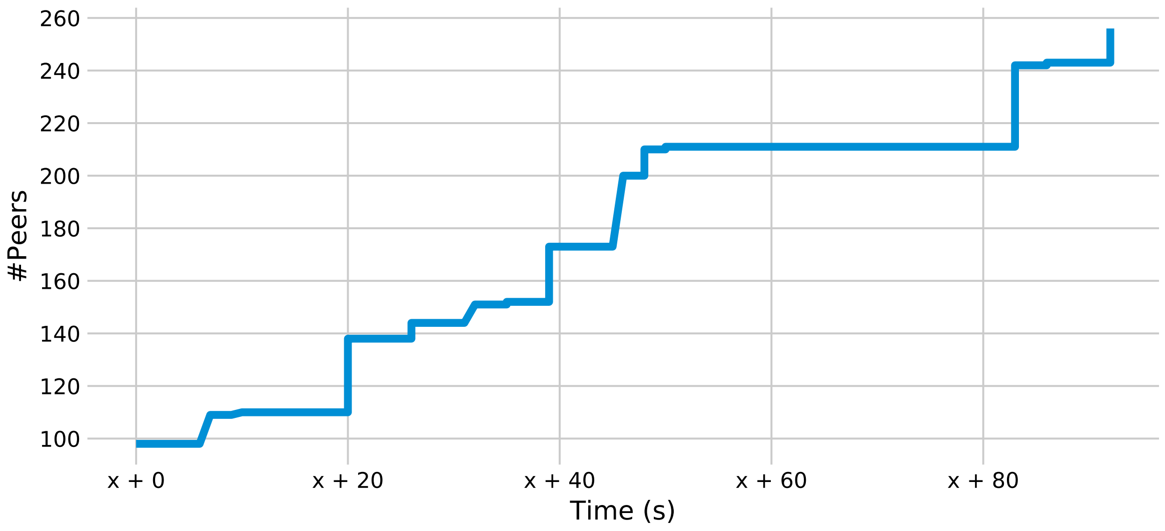


Figure 6.20: Serval Peer Discovery, Distributed Chain 256, 7 Computers

topology is distributed among all clients (x=5, one node per client), the bandwidth reduces to 437.3 Mbps and the RTTs increase to 9.8ms.

6.5.6 Serval Study

MiniWorld has already been used to simulate a *Chain 64* topology with the *Bridged LAN* network backend [4]. The distributed show case which is presented in the following, was able to run a **Chain 256** topology. For that purpose, *Test System 1* and *2* have been combined. Additionally, an unvirtualized computer with the same hardware specifications as described by *Test System 2* but with 32GiB of memory has been added. 7 computers in total, provided 384 $(5 * 16 + 32 + 256)$ GiB of RAM and 112 virtual CPU cores $(6 * 8 + 64)$. 5 of 7 computers were virtualized with KVM. The code of the experiment has already been illustrated in Section 5.8.3. On all nodes *Servald* is started sequentially. After all instances are started, the *Serval* peers of node 1 are logged. The results are depicted in Figure 6.20. X is the time required to start all *Servald* instances. It took 92s to discover all peers after Serval has been started on all nodes.
More interesting is the scheduling of nodes across the servers. The capabilities of the nodes as well as the scheduling decisions of the *NodeDistributionScore* scheduling algorithm are shown in Table 6.4. *Client* 2 is the computer from *Test System 1* and has 64 virtual cores and 256GiB of RAM, therefore it gets 120 nodes assigned since the *bogompis* value is very high. This is the computer which has been used for the non distributed experiments. The Qemu nodes are assigned 512MiB of memory, thus using $120 * 512 = 61,440MiB$ on client 2. 6 of the clients have the same hardware, but only client 3 is not virtualized with KVM which results in a slightly higher *bogompis* value. The small difference is not enough to get more nodes assigned than the virtualized clients. The last clients gets the remaining nodes assigned due to floating point inaccuracy and because only whole nodes can be scheduled.
The CPU usage of the cluster is illustrated in Figure 6.21. The second last two clients are not virtualized with KVM and thus provide better performance. *Nested virtualization* shows higher CPU values in the experiment (nodes 1-5) which is not taken into account by the scheduling algorithm. The overall CPU usage of the cluster is sufficient for the *Serval Chain*

Client	Hostname	Bogomips	Free RAM (MB)	#Nodes
1	Andro*	55870.56	15117	23
2	BigBox	**294406.4**	**225214**	**120**
3	Rechenschieber	55871.04	29605	23
4	Andro*	55870.56	15051	23
5	Andro*	55870.56	15529	23
6	Andro*	55870.56	15232	23
7	Andro*	55870.56	15135	21

Table 6.4: Cluster Resources

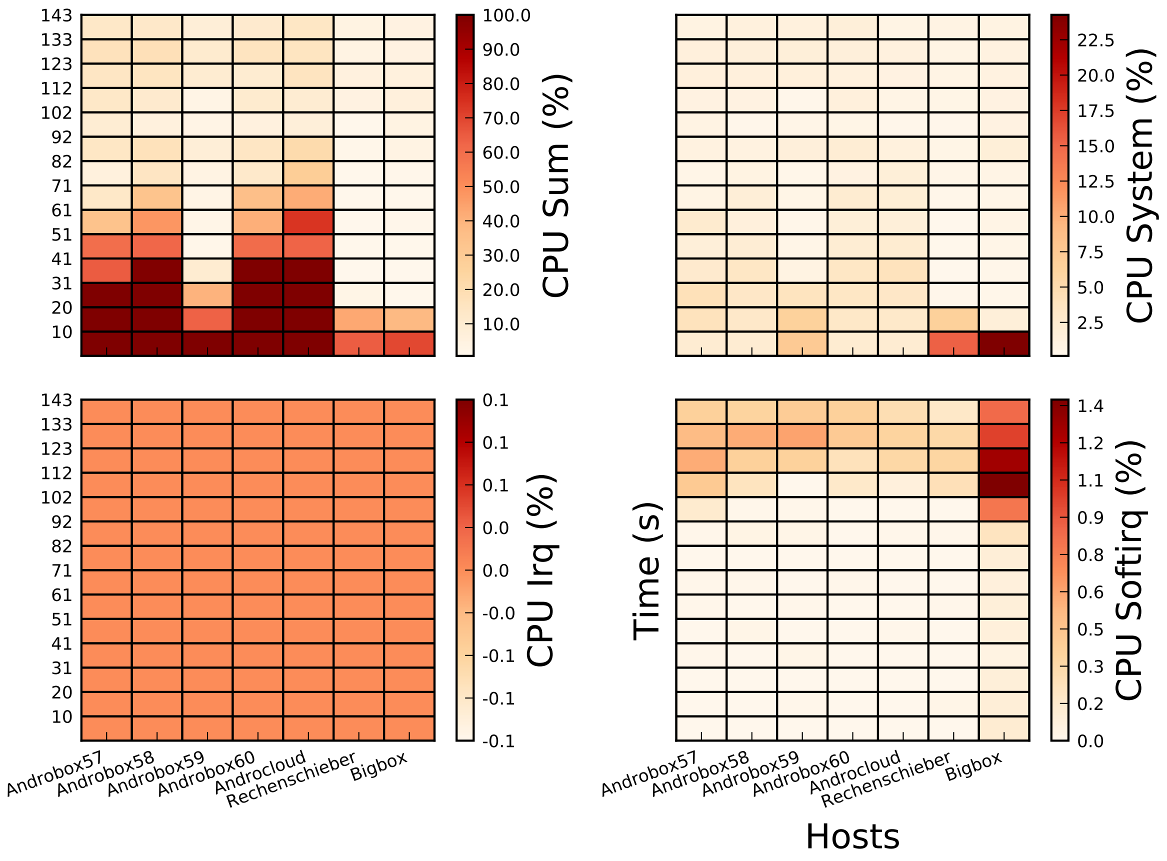


Figure 6.21: CPU Usage Serval Chain 256

256 topology at least for node discovery. The *Softirq* CPU value of the *Bigbox* client is very high. Drivers are encouraged to perform work in software interrupts such that they do not block the interrupt handler very long [47]. An explanation for the high *Softirq* value may be the high number of virtual NICs and also the number of *Serval* nodes which are trying to discover each other.
The *System* CPU value is higher for the unvirtualized nodes (*Rechenschieber*, *BigBox*) which points out that nested virtualization poses high requirements on the CPU.
Figure 6.22 depicts the CPU usage of a *512 Chain* with *Serval* which did not succeed in discovering each other. The Figure points out that the CPU usage was too high for the computers, especially the unvirtualized ones (nodes 1-5). Moreover, the *Softirq* value was

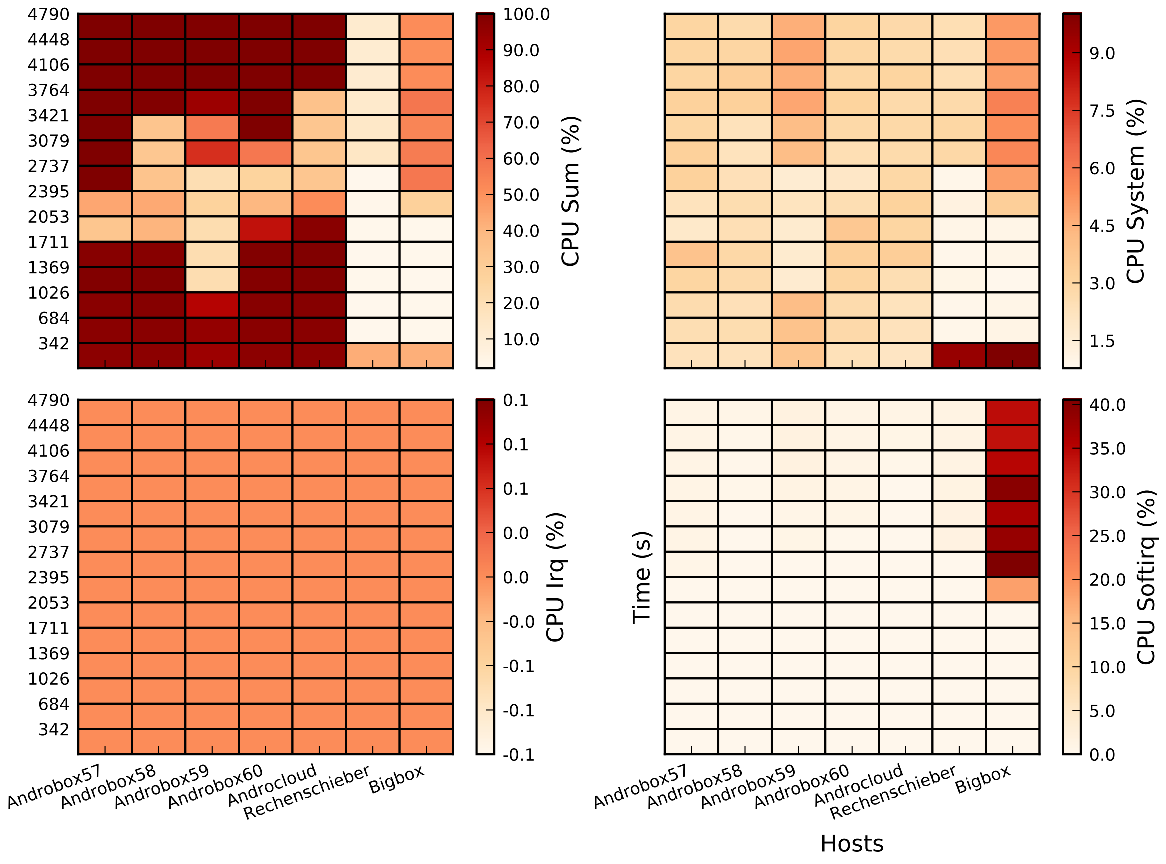


Figure 6.22: CPU Usage Serval Chain 512

very high for the *Bigbox*. The unfair scheduling of nodes has been shown in the boot times of nodes. The *Bigbox* was waiting for slower nodes, thus reducing the overall performance since all computers are kept in sync. 4790 seconds were required to only start *Servald* sequentially on all nodes.

6.6 Summary & Best Practice

The experiments outlined which settings help getting out the best performance out of MiniWorld. Choosing a VM image for an experiment depends on the properties which are important for the study: *OpenWrt* provides only the base system for router images resulting in a smaller image size and less CPU and RAM consumption. Even though other Linux distributions include a netinstall image, an OpenWrt installation is still smaller. The VM image choice is a tradeoff between performance and flexibility.

6.6.1 Boot Mode

The **Shell Prompt** boot mode shall be preferred if the user is able to place a custom signal to the end of the boot process. The *Boot Prompt* mode can be chosen too, but the user has to check whether the **Boot Prompt** is displayed late enough in the boot process. The start times of both modes should be neglected since the **Snapshot Boot** mode prevents VMs

Property	WiFi	VDE	Bridged LAN	Bridged WiFi
WiFi NIC	y	-	-	
Mobility Pattern	-	All	CORE	All
Link Quality Model*	-	1	2,3	2,3
Broadcast Domain	y	y	-	y
Interfaces: Management/Hub	-/-	y/y	y/y	y/y
Distributed Mode	?[12]	-	Gretap/VLAN/VXLan	Gretap/VLAN/VXLan
Link Impairment	-	Wirefilter	TC (HTB+Netem)	TC (HTB+Netem)
Connection Switching	-	Wirefilter	Iproute2/Pyroute2/Brctl	Iproute2/Pyroute2/Brctl

Table 6.5: Network Backend Features

Index	Link Quality Model
1	Fixed-Range
2	WiFi Simple Linear
3	WiFi Simple Linear

Table 6.6: *Link Quality Models

from being rebooted for a new experiment. Instead, VM snapshots are loaded. The user has to take care of setting the correct names for scenarios, since they are used for the snapshot names. Moreover, if the configuration of Qemu changes, a reboot is required.

6.6.2 Network Backend

Choosing the best suited network backend depends on the user requirements. Table 6.5 depicts the features of each network backend. The **WiFi** network backend is the only one which provides a wireless NICs to Linux guests, but does not provide link impairment and connection switching. All nodes are placed in a single collision domain. Moreover, experiments showed that it only provides 165.1 Mbps of bandwidth with a single CPU core. Nevertheless, it is the only option if a real WiFi NIC is required. Usage examples are 802.11s or WiFi operation modes such as AP and *Ad Hoc*.

The *VDE* network backend requires further improvement. Moreover, no connection switching experiments have been performed at the time of writing since first *VDE's* bandwidth bug has to be fixed. The **Bridged LAN** network backend is the best choice for the emulation of wired networks since packets are not broadcasted to every connected node. The limitation is that the number of connections need to be known beforehand so that only the *CORE Mobility* pattern is supported at the time of writing.
The **Bridged LAN** allows to be used with every *Mobility Pattern* and places interconnected nodes on the same collision domain, thus mimicking the wireless broadcast nature.
Network Checking should be enabled to ensure that the network is set up correctly.

[12]The WiFi network backend has not been evaluated in the distribute mode at the time of writing, but multicast routing should enable the distributed communication of nodes.

Bridged Network Backend Options

If one of the *Bridged* network backends is used, all **Execution Options** should be activated for best connection switching times. Moreover, the **Iproute2** execution option yields the best results but may require manually compiling the correct *Iproute2* version. A good compromise is the *Pyroute2* execution option, although it lacks transparency since no commands are logged. The *Brctl* execution option yields slow switching times but worked out of the box in the experiments.

6.6.3 Distributed Mode

The **NodeDistributionScore** scheduling algorithm should be chosen if the hardware resources of the clients in the distributed mode is not homogenous. Otherwise the **NodeDistributionEqual** can be chosen to achieve a uniform distribution of nodes among clients. The **Publish-Subscribe** pattern with **MessagePack** as serialization format should be preferred for client coordination since it showed the best results.

7 Conclusion

A distributed emulation framework called **MiniWorld** has been presented. The typical workflow of MiniWorld and its architecture have been illustrated in detail. MiniWorld is **modular**, **flexible** and **exchangeable** which is outlined by the integration of 4 network backends. The design incorporates adding a new virtualization layer or network backend. To overcome the performance penalties which are introduced by full system virtualization, a **Snapshot Boot** mode has been developed. Additionally, an emulation can be distributed across many computers. A special **Scheduling Algorithm** for node placement has been developed which is aware of the the critical resources which can lead to bottlenecks during an emulation.
MiniWorld offers a real WiFi NIC to its virtualized nodes. This enables MiniWorld to be used as a WMN network emulator. A use case has been presented which shows the feasibility of the **WiFi virtualization** approach. For scenarios where no wireless card is required, the **Bridged WiFi** network backend offers better bandwidth while consuming less CPU. The network backend is based on Linux bridges, ebtables and Linux TC and enables link impairments for every connection between two nodes. MiniWorld is not limited to wireless emulation: The **Bridged LAN** network backend enables the emulation of wired networks which shows the flexibility of the distributed emulation framework. A specially developed **CORE Mobility** pattern eases the development of a network backend and the testing of software since node movement is statically defined. Therefore, experimentally gathered results are reproducible. If early prototyping is finished, more advanced mobility patterns can be used instead. Since experiments require special note setup, shell commands can be defined in the *Scenario Config* to bootstrap the experiment on all nodes. Furthermore, most network backends offer a **Management Network**. The control channel can be used for experiment monitoring, further automation and experiment supervision.
The distributed *MiniWorld* emulation framework provides **Connection Tracking**, **Differential Connection Switching** and especially **Network Checking** to every network backend. In consequence, network backends are lightweight and fast. For shell based network backends, the **ShellCommandSerializer** further optimizes switching times. Processes which expose an API via a UDS can be integrated with the **REPLable** mechanism. Basic **Link Quality Models** are integrated by MiniWorld to show the feasibility of link impairments for the network backends.
The experimental evaluation outlined the bandwidth, delay and connection switching times of the 4 network backends. Moreover, node boot modes have been studied. The feasibility of the distribute mode has been presented by the **Serval** showcase.

7.1 Future Work

There are still improvements for MiniWorld: **High fidelity link emulation** and more advanced **Mobility Patterns** are required. MiniWorld could be combined with the realtime scheduler of *Ns-3* similarly to *CORE* and *Dockemu*. **WiFi virtualization** together with high

fidelity emulation is not covered by literature to the best knowledge of the author.
Link impairment can only be applied to all interface types equally. Further experiments are required to ensure that performance does not suffer from the introduced complexity, but since link impairments of the *Link Quality Models* are precomputed, no additional performance overhead can be expected.
For various scenarios, **Docker** containers could provide a lightweight alternative to KVM where full system virtualization is not required. Moreover, since software is already packages into Docker images which are available via the *Docker Hub*, it would allow to easily test distributed applications.
The **Android emulator** is built upon Qemu, hence MiniWorld could be easily extended to support Android. Android is especially interesting, since geo locations[1], can be sent to emulated phones. Currently, only distances between nodes are communicated by the **MovementDirector**, but location information could be added so that the location of Android phones could be updated by MiniWorld.
The **WiFi** network backend requires further research since no link impairment is provided at the time of writing. Moreover, all nodes are placed onto the same collision domain. The **VDE** network backend suffers from performance penalties which depends on the size of the *VDESwitch*. Further research is required to either fix the bug or circumvent it by increasing the switch size dynamically if more links are required. Moreover, links are assumed to be everlasting. Since the *VDE* network backend requires one user-space process per connection, a possibility to remove a connection from the *Connection Tracking* mechanism should be included.
Finally, since the topology generator of the CORE UI is limited depending on the topology to approximately 128 nodes, a more advanced topology generator is required.

[1] `https://stuff.mit.edu/afs/sipb/project/android/docs/tools/devices/emulator.html`. Last viewed on 06.12.2016.

Appendix

Listing 38: Scenario Config Template

```
{
  "scenario" : "scenario name",
  "cnt_nodes" : 5,

  "walk_model" : {
    "name" : "RandomWalk",
    "filepath" : "somepath"
  },

  "provisioning" : {
    "image": "images/openwrt-x86-kvm_guest-combined-ext4-batman-adv.img",
    "parallel" : true,
    "boot_mode" : "selectors",

    "regex_shell_prompt" : "(.*)root@OpenWrt.*[#]?",
    "regex_boot_completed" : "procd: - init complete -.*",
    "shell" : {
      "pre_network_start" : {
        "shell_cmd_file_path" : null,
        "shell_cmds" : [""]
      },
      "post_network_start" : {
        "shell_cmd_file_path" : null,
        "shell_cmds" : [""]
      }
    }
  },

  "qemu" : {
    "ram" : "32M",
    "qemu_user_addition": "-hdb stick.img",
    "nic" : {
      "model" : "virtio-net-pci"
    }
  },

  "network" : {
    "backend" : {
      "name" : "bridged",

      "connection_mode": "single",
      "execution_mode" : {
        "name" : "iproute2",
        "parallel" : false,
        "batch" : false,
        "one_shell_call" : false
    },
    "event_hook_script" : "path to event script",
    "distributed_mode" : "gretap",
    "tunnel_endpoints" : {},

    "num_ports" : null
  },

  "links" : {
    "miniworld.model.network.linkqualitymodels.
↪   LinkQualityModelRange.LinkQualityModelRange"

    "configuration" : {
      "auto_ipv4" : true,

      "connectivity_check" : {
        "enabled" : true,
        "timeout" : 60
      },
      "ip_provisioner" : {
        "base_network_cidr" : "10.0.0.0/8",
        "prefixlen" : 16
      },
      "nic_prefix" : "eth"
    },
    "interfaces" : ["mesh", "hubwifi"],
   "bandwidth": 55296000
  },
  "core" : {
    "topologies" : [
      [10, "MiniWorld_Images/examples/core_scenarios/no_network.xml"],
      [10, "MiniWorld_Images/examples/core_scenarios/chain_4.xml"]
      ],
    "loop" : false,
    "mode" : "lan"
    }
  },

  "distributed" : {
    "node_id_mapping" : {},
    "server_id" : 0
  },

  "node_details": {
    "1": {
      "walk_model": "RandomWalk",
      "provisioning" : {
        "shell_cmds" : ["echo miniworld"],
      },
      "qemu" : {
       "qemu_user_addition": "-hdb other_stick.img"
      }
    }
  }

}
```

Bibliography

[1] Jeff Ahrenholz. Comparison of core network emulation platforms. In *2010-milcom 2010 military communications conference*, 2010 (cited on pages 28, 30).

[2] Jeff Ahrenholz, Claudiu Danilov, Thomas R Henderson, and Jae H Kim. Core: a real-time network emulator. In *Military communications conference, 2008. milcom 2008. ieee*. IEEE, 2008, pages 1–7 (cited on pages 28, 30).

[3] Jeff Ahrenholz, Tom Goff, and Brian Adamson. Integration of the core and emane network emulators. In *Military communications conference, 2011-milcom 2011*. IEEE, 2011, pages 1870–1875 (cited on pages 29, 30).

[4] Lars Baumgärtner, Paul Gardner-Stephen, Pablo Graubner, Jeremy Lakeman, Jonas Höchst, Patrick Lampe, Nils Schmidt, Stefan Schulz, Artur Sterz, and Bernd Freisleben. An experimental evaluation of delay-tolerant networking with serval (cited on page 117).

[5] Fabrice Bellard. Qemu, a fast and portable dynamic translator. In *Usenix annual technical conference, freenix track*, 2005, pages 41–46 (cited on pages 18, 19).

[6] Davide Brini. Gre bridging, ipsec and nfqueue. URL: `http://backreference.org/2013/07/23/gre-bridging-ipsec-and-nfqueue/` (cited on page 76).

[7] José Daniel Britos, Silvia Arias, Nicolás Echániz, Guido Iribarren, Lucas Aimaretto, and Gisela Hirschfeld. Batman adv. mesh network emulator. In *Xxi congreso argentino de ciencias de la computación (junín, 2015)*, 2015 (cited on page 23).

[8] Common open research emulator (core). Last viewed on 31.10.2016. URL: `https://www.nrl.navy.mil/itd/ncs/products/core` (cited on page 27).

[9] Emmanuel Conchon, Tanguy Pérennou, Johan Garcia, and Michel Diaz. W-nine: a two-stage emulation platform for mobile and wireless systems. *Eurasip journal on wireless communications and networking*, 2010(1):1–20, 2009 (cited on page 1).

[10] Core documentation, release 4.8. 06.06.16. URL: `http://downloads.pf.itd.nrl.navy.mil/docs/core/core_manual.pdf` (cited on page 29).

[11] Renzo Davoli. Vde: virtual distributed ethernet. In *First international conference on testbeds and research infrastructures for the development of networks and communities*. IEEE, 2005, pages 213–220 (cited on page 23).

[12] Ebtables/iptables interaction on a linux-based bridge. Last viewed on 14.11.16. URL: `http://ebtables.netfilter.org/br_fw_ia/br_fw_ia.html` (cited on page 14).

[13] Michael Eder. Hypervisor-vs. container-based virtualization. *Future internet (fi) and innovative internet technologies and mobile communications (iitm)*, 1, 2016 (cited on page 19).

[14] Ramon R Fontes, Samira Afzal, Samuel HB Brito, Mateus AS Santos, and Christian Esteve Rothenberg. Mininet-wifi: emulating software-defined wireless networks. In *Network and service management (cnsm), 2015 11th international conference on*. IEEE, 2015, pages 384–389 (cited on page 31).

[15] Daniele Furlan. Analysis of the overhead of batman routing protocol in regular torus topologies. *University of trento, italy, tech. rep*, 2011 (cited on page 23).

[16] Vijay Garg. *Wireless communications & networking*. Morgan Kaufmann, 2010 (cited on page 8).

[17] Matthew Gast. 802.11 wireless networks: the definitive guide, 2002 (cited on pages 7, 8).

[18] S. Hemminger. Network emulation with netem. In *Linux conf au*, 2005. URL: `http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.67.1687&rep=rep1&type=pdf` (cited on page 22).

[19] Jerome Henry and Marcus Burton. 802.11 s mesh networking. *Cwnp whitepaper (nov. 2011)*, 2011 (cited on pages 11, 12).

[20] Guido R Hiertz, Dee Denteneer, Sebastian Max, Rakesh Taori, Javier Cardona, Lars Berlemann, and Bernhard Walke. Ieee 802.11 s: the wlan mesh standard. *Ieee wireless communications*, 17(1):104–111, 2010 (cited on pages 11, 12).

[21] Pieter Hintjens. Ømq - the guide. Last viewed on 11.11.16. URL: `http://zguide.zeromq.org/page:all` (cited on pages 53, 54).

[22] Luc Hogie, Pascal Bouvry, and Frédéric Guinand. An overview of manets simulation. *Electronic notes in theoretical computer science*, 150(1):81–101, 2006 (cited on page 21).

[23] Bert Hubert, Thomas Graf, Gregory Maxwell, Remco van Mook, Martijn van Oosterhout, Paul B Schroeder, Jasper Spaans, and Pedro Larroy. Linux advanced routing & traffic control howto. Last viewed on 28.11.2016. URL: `http://www.lartc.org/lartc.html#LARTC.QDISC` (cited on page 14).

[24] Sylvester Keil and Clemens Kolbitsch. Stateful fuzzing of wireless device drivers in an emulated environment. *Black hat japan*, 2007 (cited on page 26).

[25] Avi Kivity, Yaniv Kamay, Dor Laor, Uri Lublin, and Anthony Liguori. Kvm: the linux virtual machine monitor. In *Proceedings of the linux symposium*. Volume 1, 2007, pages 225–230 (cited on page 18).

[26] Robert Kuschnig, Evsen Yanmaz, Ingo Kofler, Bernhard Rinner, and Hermann Hellwagner. *Profiling ieee 802.11 performance on linux-based networked aerial robots*, 2012 (cited on pages 15, 16, 26).

[27] Tianyi Li, Walter E Thain Jr, and Thomas Fallon. On the use of virtualization for router network simulation. In *American society for engineering education*. American Society for Engineering Education, 2010 (cited on page 33).

[28] Emmanuel Lochin, Tanguy Perennou, and Laurent Dairaine. When should i use network emulation? *Annals of telecommunications-annales des télécommunications*, 67(5-6):247–255, 2012 (cited on pages 1, 4–6).

[29] Mac80211_hwsim. Retrieved on 05.11.2016. URL: `https://wireless.wiki.kernel.org/en/users/drivers/mac80211_hwsim` (cited on page 26).

[30] Sahibzada Ali Mahmud, Shahbaz Khan, Shoaib Khan, and Hamed Al-Raweshidy. A comparison of manets and wmns: commercial feasibility of community wireless networks and manets. In *Proceedings of the 1st international conference on access networks*. ACM, 2006, page 18 (cited on page 9).

[31] Martin Lucina Martin Sustrik. Zmq_socket. Last viewed on 22.11.2016. URL: `http://api.zeromq.org/2-1:zmq-socket` (cited on page 54).

[32] Nasa science: wave behaviors. Last viewed on 11.11.16. URL: `http://missionscience.nasa.gov/ems/03_behaviors.html` (cited on page 7).

[33] Rogério Leão Santos de Oliveira, Christiane Marie Schweitzer, Ailton Akira Shinoda, and Ligia Rodrigues Prete. Using mininet for emulation and prototyping software-defined networks. In *Communications and computing (colcom), 2014 ieee colombian conference on*. IEEE, 2014, pages 1–6 (cited on pages 23, 31).

[34] Ben Pfaff, Justin Pettit, Teemu Koponen, Ethan Jackson, Andy Zhou, Jarno Rajahalme, Jesse Gross, Alex Wang, Joe Stringer, Pravin Shelar, et al. The design and implementation of open vswitch. In *12th usenix symposium on networked systems design and implementation (nsdi 15)*, 2015, pages 117–130 (cited on page 24).

[35] Maurizio Pizzonia and Massimo Rimondini. Netkit: easy emulation of complex networks on inexpensive hardware. In *Proceedings of the 4th international conference on testbeds and research infrastructures for the development of networks & communities*. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), 2008, page 7 (cited on page 32).

[36] Matija Pužar and Thomas Plagemann. Neman: a network emulator for mobile ad-hoc networks. *Research report http://urn. nb. no/urn: nbn: no-35645*, 2005 (cited on pages 33, 34).

[37] Devan Rehunathan, S Bhatti, Vincent Perrier, and Pan Hui. The study of mobile network protocols with virtual machines. In *Proceedings of the 4th international icst conference on simulation tools and techniques*. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), 2011, pages 115–124 (cited on page 32).

[38] Jamal Hadi Salim. Linux traffic control classifier-action subsystem architecture (cited on page 14).

[39] S Salsano, F Ludovici, A Ordine, and D Giannuzzi. Definition of a general and intuitive loss model for packet networks and its implementation in the netem module in the linux kernel. *Niversity of rome "tor vergata", rome*, 2012 (cited on page 22).

[40] Andrew S Tanenbaum and Herbert Bos. *Modern operating systems*. Prentice Hall Press, 2014 (cited on pages 16–18).

[41] Andrew S Tanenbaum and David J Wetherall. Computer networks, 5-th edition. *Ed: prentice hall*, 2011 (cited on pages 3, 4, 6, 7, 9–11).

[42] Marco Antonio To, Marcos Cano, and Preng Biba. Dockemu–a network emulation tool. In *Advanced information networking and applications workshops (waina), 2015 ieee 29th international conference on*. IEEE, 2015, pages 593–598 (cited on pages 27, 30, 31).

[43] Vde. Retrieved on 07.11.2016. URL: `http://wiki.v2.cs.unibo.it/wiki/index.php?title=VDE%5C#VDE_components` (cited on page 23).

[44] M Vipin and S Srikanth. Analysis of open source drivers for ieee 802.11 wlans. In *Wireless communication and sensor computing, 2010. icwcsc 2010. international conference on*. IEEE, 2010, pages 1–5 (cited on pages 15, 26).

[45] Shie-Yuan Wang. Comparison of sdn openflow network simulator and emulators: estinet vs. mininet. In *2014 ieee symposium on computers and communications (iscc)*. IEEE, 2014, pages 1–6 (cited on page 31).

[46] Elias Weingärtner, Hendrik Vom Lehn, and Klaus Wehrle. A performance comparison of recent network simulators. In *Communications, 2009. icc'09. ieee international conference on*. IEEE, 2009, pages 1–5 (cited on page 21).

[47] Matthew Wilcox. I'll do it later: softirqs, tasklets, bottom halves, task queues, work queues and timers. In *Linux. conf. au*, 2003 (cited on page 118).

[48] Lei Xia, Sanjay Kumar, Xue Yang, Praveen Gopalakrishnan, York Liu, Sebastian Schoenberg, and Xingang Guo. Virtual wifi: bring virtualization from wired to wireless. In *Acm sigplan notices*. Volume 46. (7). ACM, 2011, pages 181–192 (cited on pages 24, 25).

[49] Yu Yu, Shashi Shah, Yasuo Tan, and Yuto Lim. End-to-end throughput evaluation of consensus tpc algorithm in multihop wireless networks. In *2015 international wireless communications and mobile computing conference (iwcmc)*. IEEE, 2015, pages 941–946 (cited on page 26).

[50] Marko Zec and Miljenko Mikuc. Operating system support for integrated network emulation in imunes. In *Workshop on operating system and architectural support for the on demand it infrastructure (1; 2004)*, 2004 (cited on page 30).

Eidesstattliche Erklärung

Ich versichere hiermit an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst, ganz oder in Teilen noch nicht als Prüfungsleistung vorgelegt und keine anderen als die angegebenen Hilfsmittel benutzt habe. Sämtliche Stellen der Arbeit, die benutzten Werken im Wortlaut oder dem Sinn nach entnommen sind, habe ich durch Quellenangaben kenntlich gemacht. Dies gilt auch für Zeichnungen, Skizzen, bildliche Darstellungen und dergleichen sowie für Quellen aus dem Internet. Bei Zuwiderhandlung gilt die Masterarbeit als nicht bestanden. Ich bin mir bewusst, dass es sich bei Plagiarismus um schweres akademisches Fehlverhalten handelt, das im Wiederholungsfall weiter sanktioniert werden kann.

München, am ____________________ ______________________________________

Datum Unterschrift