Marcus Pieska
Computer Science Department
Karlstad University

Emulating Software-Defined Small-Cell Wireless Mesh Networks Using ns-3 and Mininet
Abstract

The objective of this thesis was to create a network emulator, suitable for evaluating solutions in a small-cell wireless mesh SDN backhaul network environment, by integrating existing software. The most important efforts in this process have been a transparent integration of Mininet and ns-3 at both the data and the control plane, with ns-3 serving as the front-end. The goal has been to design the system such that solutions revolving around fast failover, resilient routing, and energy efficient small cell management may be evaluated. The constituent components include an augmented ns-3 WiFi module with millimeter wave communication capabilities; a socket API suitable for remote-controller management, as well as the network emulator Mininet. Mininet in turn integrates Open vSwitch, virtual hosts in the form of Linux network namespaces, and OpenFlow controllers. The work has also included a brief evaluation of the system, which revealed that the design has a fundamental flaw.
## Contents

1 Introduction

1.1 A Crash Course Introduction ........................................... 1
1.2 Second Project .......................................................... 3
1.3 Project Results .......................................................... 4
1.4 Outline ................................................................. 4

2 Background

2.1 Wireless Communication .................................................. 5
  2.1.1 Small cells and mmWave communications .......................... 7
  2.1.2 Small cells and energy efficiency ................................. 8
  2.1.3 Wireless simulations and ns-3 .................................... 8
2.2 Software Defined Networking ........................................... 10
  2.2.1 The OpenFlow protocol .......................................... 13
  2.2.2 Open vSwitch ..................................................... 14
  2.2.3 Emulation, SDN, and Mininet ................................. 14
2.3 Summary ................................................................. 15

3 Design

3.1 Overview ................................................................. 16
3.2 The Network Model ..................................................... 18
3.3 Data Plane Integration .................................................. 20
  3.3.1 The usage of TAP devices .................................... 20
  3.3.2 Transparent configuration of Mininet .......................... 21
3.4 Control Plane Integration ............................................... 23
  3.4.1 Different approaches .......................................... 23
  3.4.2 Altering OVS or not? ........................................ 24
3.5 Summary ................................................................. 26
4 Implementation

4.1 Overview ......................................................... 27
4.2 Realization of the Network Model ............................. 29
    4.2.1 The network model revisited ............................ 31
    4.2.2 Static and dynamic model ............................... 32
4.3 Data Plane Integration ........................................ 33
    4.3.1 Handling of system commands ......................... 37
    4.3.2 ns-3 ACK/CTS issue .................................... 37
4.4 The Control Socket API ....................................... 39
    4.4.1 Message parsing ........................................ 41
    4.4.2 Message casting ........................................ 45
    4.4.3 Message demultiplexing ................................. 46
    4.4.4 Message-to-variable mapping ......................... 47
    4.4.5 Example call .......................................... 48
    4.4.6 Current extent of the API ............................. 49

5 Evaluation ......................................................... 54

5.1 Controller network discovery ................................ 54
5.2 Tracing via socket API ....................................... 55

6 Discussion ......................................................... 60

6.1 Evaluation ..................................................... 60
6.2 Future Work ................................................... 61
6.3 Conclusion ..................................................... 62
# List of Figures

1.1 Example of cellular topology. ........................................ 2  
2.1 Example of an idealized cellular topology. .......................... 6  
2.2 Another example of small cells. ...................................... 7  
2.3 A schematic overview of TAP based integration. .................... 9  
2.4 Historical development of switching elements. ...................... 10  
2.5 Actual controllers view of network. ................................. 11  
2.6 Detailed view of an SDN network. .................................... 11  
2.7 SDN network with failover instance installed. ..................... 15  
2.8 Bidirectional forwarding detection. ................................ 15  
3.1 Detailed schematic view of the emulator design. ................... 17  
3.2 Aspirational example topology. ...................................... 18  
3.3 Control plane integration strategies. ................................ 25  
3.4 Detailed schematic view of the actual design. ..................... 26  
4.1 ns-3 directory with emulator additions. ............................ 27  
4.2 Realized aspirational mmWave example topology. .................. 31  
4.3 A UML diagram for data plane integration. ....................... 33  
4.4 A UML diagram for control plane integration. .................... 39  
4.5 Overview of the entire system. ...................................... 40  
5.1 Actual controllers view of emulated network. ..................... 55  
5.2 Channel management via controller. ................................ 56  
5.3 Channel utilization monitoring via controller. ..................... 57  
5.4 Power consumption monitoring via controller. ..................... 58  
5.5 TX/RX rate monitoring via controller. ............................. 59
List of Tables

3.1 Handling of TAP names and MAC addresses. .............................. 21
4.1 All traceable classes added to ns-3. ................................. 28
4.2 All the constituent parts of the socket API module. ................. 40
4.3 API socket address format for the socket API. ...................... 41
4.4 API message header format. ........................................ 41
4.5 Exposed MAC layer, station and queue parameters. ................... 50
4.6 Exposed physical layer and energy parameters. ....................... 51
4.7 Exposed mobility parameters. ...................................... 52
4.8 Exposed utilization parameters. .................................... 52
• ACL: Access Control List
• AP: Access Point
• API: Application Programming Interface
• ARP: Address Resolution Protocol
• BS: Base Station
• BFD: Bidirectional Forwarding Detection
• CLI: Command Line Interface
• IPC: Inter Process Communication
• LTE: Long Term Evolution
• NIC: Network Interface Card
• ns-3: Network Simulator 3
• ODL: OpenDaylight
• OF: OpenFlow
• OVS: Open vSwitch
• QoS: Quality of Service
• SDN: Software Defined Network
• STA: Station
• WiFi: Wireless Fidelity
• wmSDN: Wireless Mesh Software Defined Network
1 Introduction

Among recent trends in networking are two things of particular interest: The first trend is the rapidly increasing demand for wireless capacity[1], with the solution to this issue expected to entail the deployment of small cells[2] — more on this in Section 1.1. Moreover, fifth generation cellular networks are expected to make use of millimeter wave (mmWave) communication in order to increase the available spectrum[2][3][4]. The second trend is the emergence of Software Defined Networking (SDN) in general, and the more particular combination of SDN and wireless mesh networks[5][6].

1.1 A Crash Course Introduction

The five years that followed after 2011 saw an 18-fold increase in mobile data-traffic, with an increase of 63 % in 2016 alone[1]; similar growth is expected in the future, with the monthly mobile data traffic forecast to increase from 7 exabytes in 2016 to 49 exabytes in 2021. The actual growth in 2016 was, furthermore, seen on all continents, with the fastest increase in the developing parts of the world.

Given the aforementioned increase, mmWave technology looms as a very attractive future solution as it offers a large amount of unused wireless spectrum. Another expected solution is to increase wireless capacity is cell densification[7]: Since the capacity of a base station is shared among all the users within its cell, additional capacity may be provided by simply shrinking the cell, while at the same time also installing more of them to cover the same area. Another solution which falls under the same general category is to add small cells on top of larger cells in areas where the demand is high, as exemplified in Figure 1.1; this may well entail small cells using spectrum which is entirely disjoint from the spectrum used by traditional macro cells, both for the connections made between users and small cell base stations, as well as between small cell base stations. The network which small cells create amongst themselves is referred to as the small cell backhaul; this is exemplified
in Figure 1.1 as the connections made between the various small cells and between any small cell and a macro cell. Energy efficient management of small cells, as well as robust connectivity over the small cell backhaul, emerge as issues to be studied in this context.

SDN, or *Software-Defined Network(ing)*, is a new paradigm in networking whereby networks are programmable[8][9][10][11]. The idea behind SDN is to separate the control logic from the data plane; the control logic is removed from individual switches and instead placed in a central controller. Such a controller maintains a channel to each switch in the network, and thus possess a global view of the topology. A controller could help maintain connectivity over highly dynamic topologies such as mmWave-based small-cell backhaul networks; there would, for instance, not be any need for distributed routing algorithms which converge slowly. A controller could also monitor network utilization, and selectively power down small cells that are not required to meet the current demand, and thus provide the required capacity at a lower cost in terms of energy usage.
The goal of this work was primarily to develop an integrated system suitable for emulating small cell wireless mesh SDN backhaul. Secondarily — if time allows — the finished system was to be used to study how to leverage SDN to achieve robust connectivity over such networks. The backhaul aspect is, in truth, not central to understanding the design and implementation described in this thesis; the reader may instead bear in mind the more palatable wireless mesh SDN — or wmSDN[5][6]. The finished emulator integrates link models for WiFi[12] — available in the network simulator ns-3[13] — with virtual hosts and a software switch known as Open vSwitch (OVS)[14][15][16]. OVS is an SDN compatible software switch that is usable in real deployments; thus, real controllers are able to manage networks emulated using the finished product described in this thesis.

The following points were specified by the product owner from the very start; all issues involve leveraging SDN: (I) to achieve resilient routing; (II) for energy efficient management of small cell backhaul networks; (III) to get the fast failover capability of OpenFlow for a faster response to lost connectivity. What these three issues amount to in terms of design and implementation will be discussed in later chapters of this thesis.

1.2 Second Project

A second project, running in parallel to the one described throughout this dissertation, has as a goal to extend OVS with two interfaces. The first interface is to interact with the modeled network devices in ns-3; the second interface is to allow an SDN controller to manage ns-3 net devices via the first interface. One way to describe the contribution of this second project is to point out that without it the controller is only able to manage those aspects of the network which are general; with the second project, the controller is also able to manage those things which are particular to the ns-3 WiFi net devices. With these interfaces in place, all that is required for the system to move to a real deployment is that the first interface is exchanged for some other interface which works with actual, tangible network interfaces — or wireless radios.
1.3 Project Results

The result, as of the end of the project, is an integrated network emulator which makes use of Open vSwitch (OVS), host virtualization via Linux network namespaces, the ns-3 WiFi module, and any controller that is able to communicate with OVS. The host virtualization allows the user to use common network tools — such as *iperf*, *tcpdump*, *tcptrace*, and *ping* — via the well-defined CLI of such programs; host virtualization also result in real network stacks being used in the emulator — for instance real TCP protocols. ns-3 serves as the front-end for the system as a whole, and it has been given a module which acts as an API written specifically for defining wmSDN topologies. The WiFi module, the raison d’être for ns-3 in this context, has been given a socket API which exposes several dozen parameters; it is envisioned that a controller could eventually use this interface to perform low level management of any WiFi network device in the emulated network. A mock controller was also created for testing purposes, allowing for use of the system in lieu of the results from project described in Section 1.2.

1.4 Outline

The remainder of the thesis is organized as follows: (II) a background that covers pertinent information about SDN, wireless communication, and network emulation; (III) a design section, split into one part each for the data plane and control plane integration; (IV) a chapter on implementation, split along the same lines as the design section; (V) an evaluation, including recounting of some debugging, a broad evaluation of the system, and a proof-of-concept type testing which illustrates that the integrated system is sufficient given the previously described requirements; (VI) lastly, a conclusion that discusses the outcome of the work, as well as potential future work.
2 Background

This chapter is split into the following sections: (I) pertinent information on wireless communication, small cells, mmWave communication, and the support for such things in current network simulators; (II) a summary of SDN, OpenFlow, Open vSwitch, and current emulators that support SDN.

2.1 Wireless Communication

Chapter 1 discussed the dramatic increase in demand for wireless capacity, and the forecast increase in the near future; it also mentioned some of the expected solutions to meet the aforementioned increasing demand — these solutions include a future deployment of small cells and the use of the millimeter wave frequency domain where unused spectrum is available. Section 2.1.1 briefly expound on the subject of mmWave communication as well as the relationship between mmWave and small cells; energy efficient management of small cells and their backhaul is discussed in Section 2.1.2; simulation possibilities for wireless mesh networks is discussed in Section 2.1.3.

Figure 2.1 depicts the idealized example of a cellular network. The topology consists of a number of cells with each cell having a base station (BS) which provides network access to all the mobile users within the cell, using a particular frequency domain. Base stations in neighboring cells may not use the same frequency since they would then interfere with each other; as such, the frequency band as a whole has to be split into parts which in turn are then assigned to the base stations in such a manner as to avoid interference. Figure 2.1 illustrates several cells which make use of the same frequency area; none of these cells border each other. Implicit in Figure 2.1 is that the interference of a BS does not reach beyond its immediate neighbors — if it did, the frequency reuse distance would have to be larger than a single cell. If a larger reuse distance has to be used, the frequency domain has to be split into additional parts; thus the per-cell capacity would be reduced.
LTE is a modern cellular architecture that is in common use; a base station in LTE is referred to as an eNodeB, while a user is referred to as a UE, or user equipment. WiFi is the marketing name for several wireless LAN standards that go under the technical heading of IEEE 802.11; there are several standards within IEEE 802.11, one of which is IEEE 802.11ad which make use of millimeter wave communication. The closest concept to a base station in WiFi is the access point (AP), while users are referred to as stations (STA). These terms will be used throughout this report.

Figure 2.1 describes LTE and other cellular technologies in an idealized manner, but does not apply to WiFi; cellular networking involves carefully planned cells that make use of licensed spectrum and which are operated by large telecommunication companies. Figure 2.1 illustrates how, for instance, the frequency domain in that example is split up into three areas — one for each color of cell. In contrast, WiFi access points operate in unlicensed spectrum and are deployed in less planned way than are base stations. Indeed, there is even an ad-hoc WiFi MAC that is more akin to the peer-to-peer paradigm, than the hierarchical relationship between the access point and stations.
2.1.1 Small cells and mmWave communications

When it comes to increasing available wireless capacity, there are three paradigms; these aim to increase the \textit{used spectrum}, increase the \textit{spatial efficiency}, and increase the \textit{spectral efficiency}\cite{17} — mmWave communication entail an increase in the used spectrum. It will likely prove expensive and cumbersome to connect small cells using fiber, it is therefore likewise likely that small cell backhaul networks will instead consist of mmWave links\cite{4}.

Another option might be a backhaul that is in-band with the macro base station, but such a solution would crowd an already crowded spectrum and interfere with the macro cell. Thus, in order for traffic from a small cell to reach the Internet, it will likely have to travel over several wireless mmWave links until it reaches a base station with a fiber connection to the Internet — which will likely be the macro cell BS.

One issue is how to ensure robust connectivity over the backhaul — owing to the fact that mmWave links would make the backhaul susceptible to link outages. The reason for this mercurial behavior is the propensity of mmWave links to be blocked by fairly small obstacles; prior research has shown that even a single wall may cause signal attenuation of up to 24 db\cite{3}. This would, in practice, mean that a mmWave link that is suddenly blocked might see its throughput reduced to a mere faction of its full capacity\cite{18}\cite{19}.

![Diagram of small cell network](image)

Figure 2.2: Another example of small cells; reproduced for convenience.
2.1.2 Small cells and energy efficiency

Because the addition of small cells is a kind of brute force solution to a shortage of wireless capacity, and that small cells thus are likely to be quite numerous, the energy effective management of small cells emerges as an issue. If the two macro cells in Figure 2.2 were sufficient in terms of capacity, it would follow that the small cells should be powered down for the time being. Even if more capacity was required, some small cells could still be redundant, and thus safely powered down.

One approach is an energy effective sleep mode[20] where some modules in the small cell are completely turned off or put into a low-power state. Several ways of activating a small cell in sleep-mode have been proposed, including giving the small cell a kind of sniff capability that would allow it to detect a call even when sleeping, and thus activate itself. Another approach would be to activate the small cell using control messages from the macro base station via the backhaul. Yet another approach is to let the UE wake any sleeping small cell within its range with some kind of signaling message.

The association of users to small cells, and the subsequent routing of packets over the backhaul, affects the power consumption of that small cell backhaul. One approach to handle this issue is discussed in [21], where a heuristic is used to approach the optimal configuration of user associations and backhaul routing, aided by an SDN controller that runs the heuristic and subsequently power down redundant nodes.

2.1.3 Wireless simulations and ns-3

ns-3 is a commonly used network simulator[12][22], and the third in a series of network simulators. ns-3 is an interesting simulator because it has several features that are useful for the purposes of this thesis. There is an option in ns-3 to run the simulation in real-time, as opposed to simulated time; doing so limits ns-3 to the kind of data rates which can be processed in real-time by the system on which the simulation is executed — the more complex a link module is in ns-3, the less the throughput will be in real-time.
There is a TapBridge module in ns-3 which allow for ns-3 net devices to interface with Linux TAP devices; since a TAP device will appear on the operating system as a network interface, the TapBridge enables ns-3 net devices to appear likewise. These features allow for an interesting kind of TAP integration, as in Figure 2.3, where ns-3 is illustrated in blue while system code is in red; the upper case integration is between system endpoints; the lower is between a system endpoint and an ns-3 node. Moreover, the packet and protocol models in ns-3 are advanced enough so that ns-3 packets pass for real packets, and may thus be processed by real routers.

Lastly, ns-3 also has link models for both LTE and WiFi, and there are custom modules written by third parties for mmWave LTE[23], and mmWave WiFi[13]. As an aside, there is some support for SDN and OpenFlow version 1.3 in ns-3[24].
2.2 Software Defined Networking

As was mentioned in Chapter 1, SDN describes a new paradigm in networking in which networks are programmable[8][9][10][11]. The key distinctions of SDN, when compared to a more traditional network, include the controller which manage the network and the comparatively simple switching elements. This section will include a brief summary of the history of SDN, as well as a handful of potential benefits of SDN; Section 2.2.1 will elaborate on OpenFlow, which may be thought of as a partial realization of SDN; Section 2.2.2 will introduce Open vSwitch, a software switch that speaks OpenFlow; lastly, Section 2.2.3 details some possibilities when it comes to emulating SDN.

Figure 2.4: Historical development of switching elements[9].

Prior to the emergence of the Internet, routers tended to be software programs running on general-purpose UNIX systems[9], where software was used to carry out packet inspections and forwarding of packets based on header information. It is, in general, faster to execute an action in hardware than in software; as the Internet evolved and data rates increased, more and more of the work done by routers had to be carried out by hardware. Eventually, the hardware itself would evolve to become programmable; such programmable hardware is a prerequisite for SDN. Figure 2.4 summarizes how routers have evolved over time. OpenFlow would make its first appearance in 2008, with the term SDN appearing one year later — although it would take until around 2011 for the term to become commonly used throughout the industry.
Figure 2.5: Actual controllers view of network.

Figure 2.6: Detailed view of SDN an network.
One looked-for advantage with SDN is the simplification of network equipment; this simplification stems from the fact that SDN switches are, effectively, lobotomized, with control and management moved to the central controller. By placing the higher intelligence in a controller, it is hoped that more of an open source tradition might emerge in networking; this would increase reuse of software and decrease costs related to development. The current situation in networking is that vendors have complete control over their network equipment in the sense that only the vendors themselves may write software that manages their equipment; with SDN this would no longer be the case, and any one could write management software for an SDN controller, which would hopefully entail faster innovation. For an innovation in network management to see deployment in an SDN context all that is required is new software in controllers, whereas it might well require the replacement of network equipment in the more traditional non-SDN context.

Naturally, in terms of network performance, many potential advantages stem from the controller employed to manage software defined networks. Figure 2.5 illustrates the view of a network as captured by an SDN controller; in this case, the controller used was OpenDaylight (ODL)[25][26]. As was mentioned in Chapter 1, such a global view removes the need for distributed routing protocols with slowly converging routing tables throughout the network; instead, the entire network may learn of a change in network state at more or less the same time by being informed by the controller.

Figure 2.6 further illustrates the SDN architecture in some more detail; the key points include the two interfaces of the controller, one being the northbound interface which interacts with management applications, the other being the southbound interface which talks with the network nodes. The latter interface is of particular interest since there is a well-known and commonly used protocol that defines this interface, namely OpenFlow; Section 2.2.1 briefly elucidates this particular implementation of SDN. Note in Figure 2.6 that the southbound interface is used to setup a control channel between the controller and each node in the network.
2.2.1 The OpenFlow protocol

The previous discussion focused on the fundamental concepts behind SDN; OpenFlow is a concrete partial realization of SDN as it implements a protocol for communication over the southbound API[27][28] — as illustrated in Figure 2.6. OpenFlow is a protocol, much like TCP and IP are protocols, in that it defines a language with which a controller may communicate with the forwarding elements it purports to control. The following describes the basic operating procedure of OpenFlow: (I) the controller installs forwarding rules on the nodes under its control, according to its global view; (II) when a packet arrives at a switch, the switch attempts to match the packet against the installed rules, if a match is made, a corresponding action is carried out; (III) if no match is made, the switch instead forwards the packet to the controller; (IV) the controller, in turn, uses its greater "intelligence" and global view to determine a proper action, e.g., installing new rules in switches such that they may handle similar packets in the future. OpenFlow allows for matching against fields in the MAC, network, and transport layer headers; matching may also be done against the ingress port of the packet.

One aspect which is of particular interest is the fast-failover capacity of OpenFlow. Fast-failover may be viewed as an ordering of ports for a given match in the forwarding table, in conjunction with link failure detection. As an example, assume that a match has been made in the forwarding table, and that the first and second choice of outgoing port is ports 1 and 2, respectively; forwarding will be over port 1 as long connectivity is seen over that port; as soon as this is not the case port 2 is used instead. This allows for a faster response to link failures. The response time of a controller may make it a bad manager of wireless mesh networks[29]; arguably, a controller should only manage a wireless mesh if an abstraction is employed which hides the low-level management, allowing the distributed algorithms in the constituent nodes to handle this management instead. Another way to speed up the response to topology changes in an OpenFlow wireless mesh network is to leverage the aforementioned fast-failover capacity of OpenFlow.
2.2.2 Open vSwitch

Open vSwitch (OVS)[14][15] is a popular OpenFlow-compatible virtual software switch. One fortunate thing for the purpose of this thesis is the fact OVS has already been integrated into network emulators such as Mininet[30][31] and CORE[32][33]. A claim was made in Section 1 that OVS can be controlled by any controller; this is not necessarily true; OVS understands the OpenFlow protocol, and as such it may at least be controlled by any entity which speaks this language.

Figure 2.7 depicts the fast-failover feature mentioned in Section 2.2.1; a switch which supports OpenFlow of a sufficiently recent version should have this capability. Using the fast-failover feature entails some manner of link-failure detection, which in this case is *bidirectional forwarding detection* (BFD)[34] — Figure 2.8. BFD operates by sending control packets over the link in question, in one of several different modes; the *asynchronous* mode will have an endpoint transmit control packets at regular intervals, and if a sufficient number of these in a row fail to arrive at the other end, the link is declared down. Although there are other modes, these are omitted for the sake of brevity.

2.2.3 Emulation, SDN, and Mininet

CORE is a network emulator that allows for real transport and network-layer protocols to be used when running an emulation, while using simple models to implement the link and physical layers. CORE allows for both wired and wireless links, although the models used for these are simple.

Mininet[30][31] is network emulator similar to CORE; Mininet supports SDN by default, including the use of OVS as forwarding elements. The emulator also uses lightly virtualized hosts based on network namespaces[35][36], meaning that each host is different in that it has a separate network stack, while sharing the same file system with other hosts. Although both emulators could be used for this thesis, Mininet was chosen because its seeming simplicity and support for SDN. There is also support for WiFi in Mininet [37].
2.3 Summary

For the purposes of this thesis it suffices to know that energy efficient management of small cell backhaul networks is an issue, and that if a controller is to manage a backhaul, steps would have to be taken to speed up reaction times. The most salient parts of this background, however, revolve around the possibilities of emulating wmSDN networks; one can easily simulate wireless mesh networks using ns-3, and one could likewise easily emulate SDN using Mininet or CORE, but to get at the combination of both an integration of Mininet/CORE and ns-3 might be required. Chapter 3 picks up on this point.
3 Design

This chapter discusses the design process behind the development of the integrated emulator. The design process was, in truth, *iterative* — making splitting the report into a design section and and implementation section somewhat awkward and arbitrary. The design and implementation was primarily split into two large iterations, one iteration saw the design and implementation of the data plane integration between OVS and ns-3; the other iteration corresponded to the control plane integration between OVS and ns-3. Thus, many design details were established quite early on, especially in each of the aforementioned steps; furthermore, a design section does serve a nice soft introduction to the work done. This chapter is split into the following sections: an overview in Section 3.1; a discussion on the network model in Section 3.2; an accounting of the *data plane integration* in Section 3.3; lastly, a description of the *control plane integration* in Section 3.4.

3.1 Overview

A brief summary of the very first conception of the emulator would be the following: ns-3 was to be used to augment Mininet with more complex link models — i.e., ns-3 will add links to the overall system. One way to phrase this is to say that Mininet only instantiates the nodes in the graph, leaving ns-3 to handle the edges. It will be seen later that there are exceptions to this rule in the actual implementation. TAP devices hold Mininet nodes and ns-3 edges together; thus, whenever TAP devices are discussed one may imagine it as glue between nodes and edges. ns-3 contains models for several different link types, such as WiFi and LTE, and one may in theory use any one of these as links between Mininet nodes. Mininet is, in turn, able to setup OpenFlow-compatible OVS nodes. Lastly, since the context is SDN, a controller will want to manage each individual ns-3 net device; thus there has to be a way of passing data back and forth between such a controller and each ns-3 net device attached to an OVS.
The most abstract overview of the system sees the system as two components, i.e., two programs, which interact to emulate a wmSDN — these components being ns-3 and Mininet. However, Mininet is a network emulator which itself integrates several different programs and operating system devices, based on a user written topology script. Mininet instantiates lightly virtualized hosts and simple links, while it at the same time also configures OVS nodes and connect these to some controller. This view of the system therefore instead sees five components. Mininet is not strictly speaking required to setup any of the above components, but it is likely easier to handle the constituent parts of Mininet via Mininet, rather than individually via systems commands.

Figure 3.1 illustrates the final design, and the ultimate outcome of the implementation. If one were to adopt the first view of the system, the brown parts would be seen as Mininet while the green parts would be seen as ns-3. The grey components are features of the Linux operating system, and serve as glue between the ns-3 and Mininet. This design emerged in two separate stages; (I) the first design stage yielded the TAP-based data plane integration; (II) the second stage instead produced the UNIX-domain socket based control plane integration — or at least the ns-3 based half of it. Note that the example to the left in 3.1 is the hypothetical result of a second master thesis on top of this one, whereas the example to the right is the outcome of this master thesis. The remaining part of this chapter will be split according to these two stages, as well as an introductory segment which discusses how the system actually models a wireless mesh SDN.
3.2 The Network Model

As was mentioned in Chapter 1, the stated goal of the network emulator is that it should be able to emulate wireless mesh SDN topologies; as before, the small cell backhaul part is not central to follow the content below. The adopted model is fulfilled with the following points: (I) there should be a shared wireless ns-3 transmission medium, which the reader may think of as something of a canvas; (II) to this canvas switching elements may be added at specified coordinates; (III) a switching element could in turn have any number of radios attached to it — these should ideally be mmWave WiFi net devices operating in ad-hoc mode; (IV) lastly, any switch could act as a gateway for any number of Mininet hosts. Using Mininet hosts rather than ns-3 hosts allow for real network diagnostic tools to be used, and cause the emulator to use real network stacks — but there is no reason why support for ns-3 host could not be included. These are the things which the emulator should be able to create via the scripting frontend.
An earlier version of the model described above had a slightly different third step. The third step above entails adding *single* radios to switches; the original third step instead had the user add *radio pairs* to different nodes which would constitute an implicitly static link at run-time. Previous examples of the same type of TAP integration created implicitly wired links, and so added TAPs pairwise; work had to be done to move away from this paradigm. The model which was ultimately adopted may be thought of as having dynamic topologies; when radios are added pairwise, it is implied that the topology is to be static, according to whatever the pairing is.

Figure 3.2 illustrates a topology based on this dynamic model. Green indicates an ns-3 component, with the green box illustrating a shared wireless transmission medium; any radio in this shared medium could, potentially, successfully transmit to any other radio, depending on e.g. distances and channel assignments. Brown indicates Mininet components. The Figure depicts the shared ns-3 medium, and the larger model which also includes hosts. There is a slight subtlety in Figure 3.2: The figure does not depict the constituent components of the system to be, it rather represents an abstract visual depiction of what the scripting front-end must be able to express. Note that the model does not include a controller; this is because it was not realized early on that the controller would be something which would be handled via the scripting frontend.
3.3 Data Plane Integration

It was explicit in the very first specification, and even in earlier discussions, that Linux TAP devices could and should be used to integrate ns-3 and Mininet — or ns-3 and OVS as the case actually is; Section 3.3.1 will discuss this aspect of the design in detail. The general idea is to integrate ns-3 with Mininet, with one serving as a wrapper around the other; thus the wrapper would hide the use of the second component. Another issue is therefore how to transparently configure one component from the API of the other; the reader may think of this issue as how to make this system reasonably user friendly. This will be discussed in Section 3.3.2.

3.3.1 The usage of TAP devices

Some naming conventions are required for the TAP devices used to connect ns-3 net devices to OVS nodes. The basis for the convention opted for is the way Mininet handles node names; each switch should have the prefix $s$, to which an index is appended; each host instead uses the prefix $h$ while likewise appending an index. TAP devices will be named using the switch or host name and a TAP index, with dash separating these two; e.g., if switch $s3$ has two net devices, these would be named $s3-r1$ and $s3-r2$. One advantage of this convention is that it may be determined to which OVS node any given TAP belongs to by simply inspecting the TAP name.

It is worth mentioning that the first static model — mentioned in Section 3.2 — instead named any TAP device according to the switch or host to which it was attached, followed by a dash and the name of the node at the other end. As an example, if switch $s2$ had a device attached to it which was supposed to transmit to host $h1$, the tap over which the data would have flown would have been named $s2-h1$; this had the added benefit of making the entire topology, and not just the TAP ownership, implicit from a list of tap names. Of course, the latter convention makes no sense when links may be created dynamically; it will even become confusing after the first dynamic alteration of the topology.
Table 3.1: Handling of TAP names and MAC addresses.

<table>
<thead>
<tr>
<th>Part 1</th>
<th>Part 2</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>Node name (s1)</td>
<td>Device index (3)</td>
<td>TAP name (s1-r3)</td>
</tr>
<tr>
<td>Node name (s4)</td>
<td>Receiving node name (h1)</td>
<td>TAP name (s4-h1)</td>
</tr>
<tr>
<td>Tap owner (s2)</td>
<td>Device index (5)</td>
<td>MAC (00:00:00:02:05:00)</td>
</tr>
<tr>
<td>IP address (10.0.0.2)</td>
<td>-</td>
<td>MAC (00:00:00:00:00:02)</td>
</tr>
</tbody>
</table>

As is discussed in Chapter 4, the TAP-naming convention also suggest a solution to another problem, namely the issue of altering the receive MAC address header field to match the MAC address of the receiving ns-3 net device. Since the name of each TAP is unique, a simple mapping is used to map the TAP name onto a MAC address; this mapping allows for predictable MAC addresses for each port in the emulated topology, which is very helpful if static forwarding rules are to be used. A second mapping is used between Mininet host IP addresses and the host MAC address — this mapping makes it straightforward to setup ARP tables in each host. Table 3.1 shows how TAP names are derived in the dynamic model, as well as the defunct static model discussed previously; note that the static mapping is illustrated on the second row. Table 3.1 also illustrates how MAC addresses are derived from both TAP names and IP addresses.

### 3.3.2 Transparent configuration of Mininet

A design goal is to have ns-3 configure Mininet based on what the user writes in an ns-3 topology script, or vice versa. In other words, one component is to be given an API which also acts as a wrapper around the second component, such that the second component is used more or less transparently. The mode of this configuration is IPC via text file — or text files. Using text files is an attractive solution because of its simplicity; while it may be ineffective it is only used once during the start-up, and so the inefficiency does not hurt the performance of the system during emulation time. As will be seen in Section 4.4, the integration of the control plane requires a different solution.
One good metric to use when deciding which component is to act as the wrapper is: Which component will require the least configuring? If a component requires a lot of configuration, the API designed for this purpose becomes correspondingly larger. The complexity of ns-3 suggests that it would be difficult to create a Mininet API which wraps around ns-3; unless one opts for limiting the number of parameters exposed via this API. However, even just the ns-3 WiFi module contains an enormous number of parameters. Recall from previous discussion in Section 3.1 that Mininet may be seen as a collection of programs, such as OVS and whichever controller is used. Although these are just as complex as ns-3, it is not expected that these should be configured manually in Mininet, and thus not via ns-3 either. The controller will run as a separate entity; writing applications which manage the network and solves issues such as those listed in Chapter 1 is the whole purpose of this emulator. It is therefore probably safe to say that the controller need not be taken into consideration when deliberating this design decision — the controller will sort itself out. It is also worth keeping in mind that OVS has a well-established API, wherefore it should be easy to configure it from another program by executing system commands.

The conclusion is that ns-3 is to act as the front-end, and that it would suffice to pass three lists to Mininet via text files, these lists contain the following: (I) a list of switches to be created; (II) a list of TAP devices to be created and attached to a listed switch; (III) a list of hosts-switch pairs. Recall from the discussion in Section 3.3.1 that it need not be specified to which switch any given TAP is to be attached, this is implicit from the TAP name. In summary, what is needed is an ns-3 front-end which can express topologies such as the one exemplified in Figure 3.2. The last three steps of the four steps enumerated in Section 3.2 can then be translated into the three lists mentioned above. These three lists will then have to be written to files, after which ns-3 may start Mininet; upon start-up, Mininet will have to read and parse these three lists and act accordingly — what such action would entail is detailed in Chapter 4.4.
3.4 Control Plane Integration

Given the scenario described in Section 3.1, it is clear that low layer information will have to be transferred, somehow, from the ns-3 link module and the network device therein, to the OVS node to which the device is supposed to be attached. This is somewhat unfortunate, since this entails some kind of inter process communication (IPC), owing to the fact that one process models the links while another one handles OVS; this section will elaborate on the design of this IPC.

While the potential control traffic is perhaps not a large amount in bulk, it is certainly large in kind; i.e., the set of data in which the controller is interested in is potentially very large. The data is furthermore scattered around in several different classes, as this is the nature of the ns-3 WiFi module. As such the problem may be summarized as: A process is interested in retrieving and setting parameters, which are scattered in instances of several different classes, belonging to a different process. Furthermore, it is desirable for the IPC solution to be easily applied to different link modules — although this chapter will assume integration with the WiFi module in ns-3. At the time when this design was worked out, it was not understood that the mmWave WiFi module would in-fact not be a module of its own, but rather an extension to the WiFi module.

3.4.1 Different approaches

There are many possible approaches to implementing IPC; these include IPC via text file, named pipe, shared memory, and socket. Recall from the previous discussion in Section 3.3.2 that the IPC during configuration time between ns-3 and Mininet is via text file; this option is deemed appropriate as the amount of data to be exchanged in that case is very low, and because the exchange takes place at the start of the execution only, and not throughout it. When data is to be exchanged continuously, a faster solution becomes desirable. Thus, the solution opted for here is based on using sockets, and since the communication is within a single host, UNIX domain sockets are used.
At the ns-3 end, the idea is to somehow extend the capabilities of the net device with a socket interface, to which queries and instructions can be passed, and from which replies may flow. One approach is simply to extend any net device class for which we desire this capability; however, for reasons of portability between link types, the IPC logic is stored in a separate module. The choice is to let this new module implement virtual functions in a class which return generic answers to all queries, and which blocks all set-calls. It would then be up to the link module which requires the interface to inherit the aforementioned class, and implement those get- and set-functions which are deemed necessary and/or appropriate for the given link module. Basically, the socket interface will selectively reach all the way deep into which ever link module is used, in this case chiefly WiFi.

The earliest design had the ns-3 net device push data onto a repository, from which the data would in turn be pulled by a third party — either a controller or an intermediary such as an OVS node. The repository could have consisted of several virtual repositories, more specifically one for each WiFi module; or else several separate repositories might have been used, preferably one per ns-3 net device. This design was deemed lacking for two reasons; it does not, by itself, address how parameters may be changed in the link. It is also potentially bad for performance reasons, since it would entail an additional running process and continuous transfer of data. A pull strategy addresses many of these concerns, and so that is the approach used in this paper.

### 3.4.2 Altering OVS or not?

At the Mininet end of the control-plane integration, OVS will have to either be extended such that it can formulate queries and instructions and thereby communicate with the ns-3 net device over a socket, or else the OVS node could simply forward queries/instructions from the controller all the way to the data plane. An early draft of the design had the OVS operating according to the latter paradigm, which has the advantage of being a much simpler solution — see Figure 3.3 for an overview of this design. This solution entails using...
normal TAP devices instead of UNIX domain sockets, and the OVS will require one extra TAP per existing TAP; i.e., for every TAP over which data is forwarded to some ns-3 net device, another TAP will be required to forward control messages, over the data plane, to the WiFi module in ns-3. For this to work the forwarding fabric would have to contain one extra rule for every ns-3 net device, and there would have to be some way of distinguishing between real data packets and control packets in the data plane. Suppose, for instance, if a small set of IP addresses could be reserved for the set of devices which an OVS node may have; in this case control messages could be separated from data packets by the IP address. Note that this approach leaves the OVS source code unaltered, which makes for a far simpler solution. However, since the extra control tap devices are in the data plane, all ns-3 net devices could be perceived by the controller as extra ports leading to the network; hence the controller, or controller application, would require some extra logic allowing it to differentiate between data TAPs and control TAPs.

Ultimately, however, the first approach was adopted; this means moving the control path away from the data plane, as such the controller does not risk mistaking a control interface for a normal interface and the augmentation to the controller, which was mentioned earlier, is preempted. Compared to the solution latter the number of TAPs required is halved, but sockets are used instead. Figure 3.3 depicts the solution used to the left, with the alternative solution to the right.

Figure 3.3: Control plane integration strategies.
3.5 Summary

This chapter has discussed the design of a control and data plane integration between OVS and ns-3, with the premiere details being the use of IPC via text file for the former and UNIX socket for the latter. Chapter 4 will pick up on these two integration efforts and describe the implementation in great detail. Figure 3.3 is a reproduction of half of Figure 3.1 and depicts the design actually used as basis in Chapter 4 — excluding the second project mentioned in Chapter 1.2.

Figure 3.4: A schematic view of the actual emulator.
4 Implementation

This chapter mirrors Chapter 3 in its outline and contains the following sections: (I) an overview; (II) a look at the implementation of the scripting front-end, and how it enables a user to express the kind of topologies exemplified in Figure 3.2; (III) a section discussing the implementation of the data-plane integration and the transparent management of Mininet; (IV) a section discussing the implementation of the control-plane integration. Figure 4.1 is a useful reference to have in mind during the rest of this chapter, as it is an overview of sorts; the subsequent discussion will return the various folders and files in Figure 4.1.

![Figure 4.1: The ns-3 directory; folders are gray; files are green.](image)

4.1 Overview

The implementation discussed throughout this chapter revolves around three new ns-3 modules, as well as some minor alterations made elsewhere in the ns-3 source code that turned out to be necessary. The first module handles Mininet and the data plane integration of OVS and ns-3, while the second module was written for convenience and deals with setting up WiFi devices in a more user friendly manner — these two constitute the front-end in ns-3. Third and last is a module that implements the socket interface through which a controller may monitor ns-3 net devices.
ns-3 includes some helpful features which are highly useful during development and use of the integrated emulator. One such feature is the debug tracing functions used throughout ns-3 classes, including those implemented as part of this project. Listing 1 shows how this debug tracing may be activated with the function call `LogComponentEnable()`; in this case tracing two classes, named `ConfigDatabase` and `WifiHandler`. This trace function is only available if ns-3 has been built to include debug features; an optimized build will remove these features so as to speed up the simulation. Some added classes use normal print statements instead, thus making the trace output occur regardless of build or configuration; the rationale behind this is that it is desirable to see how the simulator is configured for any given session. More precisely, some classes that handle topology creation will print information to `stdout` every time — although this only occurs during the startup of any session, and not throughout. Good general practice would be to redirect `stdout` to a file when a simulation is launched, and to thereby create automated documentation of the configuration process. Table 4.1 enumerates all the added classes which may be traced via the ns-3 tracing feature.

Table 4.1: All traceable classes added to ns-3.

<table>
<thead>
<tr>
<th>Class</th>
<th>Handler</th>
</tr>
</thead>
<tbody>
<tr>
<td>CtrlIntf</td>
<td>TapHandler</td>
</tr>
<tr>
<td>IntfSwitch</td>
<td>NodeHandler</td>
</tr>
<tr>
<td>IntfWifi</td>
<td>MininetHandler</td>
</tr>
<tr>
<td>InnerLinkIntf</td>
<td>WifiNetwork</td>
</tr>
<tr>
<td>FileHandler</td>
<td>WifiHandler</td>
</tr>
</tbody>
</table>

The script shown in Listing 1 is consistent with many other examples in this report; all the example output generated by ns-3 to configure Mininet throughout Section 4.3 was generated by running the same script as in Listing 1. This script will create a simple topology consisting of two OVS nodes, with each OVS having a single WiFi net device attached to it, while each OVS in turn also acts as a gateway for a single host. Traffic is generated between the two hosts using Iperf.
4.2 Realization of the Network Model

Recall the model discussed in Section 3.2, this model was an abstract way of illustrating what kind of topologies the front-end should be able to express. Given that ns-3 has been extended with a module which can handle the interaction with Mininet (see Section 4.3), it turned out to be highly convenient to also create a user friendly scripting API. This front-end amounts to a new ns-3 class named WifiNetwork; this class is a helper class that sets up WiFi net devices which it then adds to the MininetHandler singleton class.

There are six basic steps to creating a simple WiFi mesh: (I) declare the WiFi network object — line 14; (II) declare all the OVS nodes which are to be part of the wireless mesh network — lines 21 and 22; (III) request hosts from Mininet — lines 24 and 25; (IV) add devices to the previously declared OVS nodes — lines 27 and 28; (V) wire each host to a switch — lines 30 and 31; (VI) start Mininet — line 36; In addition, listing 1 also illustrates how commands are stored to be executed later with lines 33 and 34.

The declaration of a node includes its position, which will be used later for each device added to it. Thus, each device belonging to the same node is guaranteed to be positioned in a consistent manner. Note that each device is assigned to a node via the first string parameter, which must match the name of an OVS node. This call takes three mandatory parameters: the name, the MAC layer type, and the rate manager. After these three parameters, optional string parameters may be passed, the format being <Parameter>=<Value>; these strings will be parsed and the right hand side will be used to identify the parameter to set, while the left hand side will be cast to the appropriate type and thence assigned. Wiring a host entails passing the host and switch names as parameters, with the bandwidth and latency of the link having default parameter values. When Mininet is started, all information is written to a file, after which Mininet may parse the data. Most importantly, all the OVS nodes declared in step two are written to a file; Mininet then reads this file and actually creates the corresponding nodes on the local operating system.
```cpp
int main (int argc , char *argv [])
{
    string varValue = "ns3::RealtimeSimulatorImpl" ;
    string variable = "SimulatorImplementationType" ;
    GlobalValue::Bind (variable , StringValue (varValue)) ;

    string variable = "true" ;
    string variable = "ChecksumEnabled" ;
    GlobalValue::Bind (variable , StringValue (varValue)) ;

    LogComponentEnable ("ConfigDatabase", LOG::LEVEL_ALL);
    LogComponentEnable ("WifiHandler", LOG::LEVEL_ALL);

    WifiNetwork wifi;
    MininetHandler* mn = MininetHandler::Get () ;
    mn->CreateArpTables (true);
    mn->ControllerToUse ("None");
    mn->FwdScriptToExecute ("single-link-iso");
    mn->AddNode ("s1", 10, 10);
    mn->AddNode ("s2", 20, 10);
    mn->AddNode ("h1", 10, 10);
    mn->AddNode ("h2", 20, 10);
    wifi.AddDevice ("s1", "STA", "C", "ChannelNumber=1");
    wifi.AddDevice ("s2", "AP", "C", "ChannelNumber=1");
    mn->Wire ("h1", "s1");
    mn->Wire ("h2", "s2");
    mn->AddCommand ("h2", "iperf -s", 5);
    mn->AddCommand ("h1", "iperf -c 10.0.0.2 -t240", 10);
    mn->StartMininet () ;

    Simulator::Stop (Seconds (60*60*24));
    Simulator::Run ();
    Simulator::Destroy ();
    return 0;
}
```

Listing 1: An example of an ns-3 script which creates a very simple wmSDN.
4.2.1 The network model revisited

Recall the model discussed in Section 3.2; it was suggested that all the radios in the Figure 3.2 were operating in ad-hoc mode, which would be consistent with the fact that the emulator is supposed to model wmSDN topologies. However, there is no programmatic reason why a radio could not be configured to run in STA or AP mode; indeed, the newly developed mmWave WiFi module[13] turned out to lack an ad-hoc mode, thus limiting this emulator to either ad-hoc mode or mmWave WiFi, but not both. Figure 4.2 illustrates a scenario in where a set of access points and stations are leveraged to mock-up a mesh network. The figure also illustrates an impossible link: Suppose that the link between S2 and S3 turned out to be redundant, and that the controller instead wished to link S2 and S1, this link would be impossible in this particular scenario because both radios happen to be stations. Thus, one would have to plan ahead and make sure that all desired couplings for the duration of a simulation are possible — in this case S1 needs another AP.
4.2.2 Static and dynamic model

Recall the previous discussion in Section 3.2 on the network model used, and the difference between a static and a dynamic model. The first model, which was static, involved the use of a kind of dummy link in Mininet, which was subsequently ignored at run time in favor of ns-3 based connectivity. This hack thereby circumvented Mininet in order that the desired integration with ns-3 might be achieved. This solution was dropped when it was decided that the model should be dynamic.

Since the previously described hack entailed adding radios in ns-3 in a pairwise manner, it was decided that Mininet should be omitted from the linking process entirely; instead, the tack was to manually connect the OVS nodes with the desired TAP devices. Since TAP devices appear as network interfaces to the operating system, TAP devices may be added as such to any existing OVS via the execution of system commands; as seen in listing 2. The obvious disadvantage of this solution is that the system will stop working if Mininet is configured to use any other software switch — note that Mininet is designed to be able to work with a number of different software switches. However, this drawback is diminished if OVS is the preferred technology — as is certainly is in the case of this thesis. On the other hand, the solution finally opted for is cleaner, and it allows for the system to adhere to the requested model. Note, however, that when using the mmWave module, the fact that one is limited to mocking up an ad-hoc network using access points and stations forces the model to become a kind of semi-static or semi-dynamic model.
4.3 Data Plane Integration

FileHandler handles the file IO required according to the design discussed in Section 3.3.2. The class may be thought of as a wrapper around an instance of a C++ string and some file-stream logic. The constructor accepts a directory and a filename as parameters. Using the class entails appending lines to an instance of it; once the content is completed, the owner makes a call to commit the content to a file, according to the directory and filename passed to the instance during construction. With the content stored on the hard-drive, a separate process may want to read file and parse the file, or the file may be executed as a script — the permission rights of the file are set to allow all these things. Alternatively, FileHandler allows for the ns-3 owner to directly execute the content as a command.
According to the design expounded upon in 3.3.2, configuring Mininet via ns-3 requires ns-3 to accept a configuration via its own front-end and write this configuration to files in the form of lists and scripts. A class named TapHandler handles part of this configuration; this class leverages three instances of FileHandler to create three different scripts containing system commands that need to be executed for the purpose of: (I) adding taps; (II) attaching taps previously created to OVS nodes — as ports; (III) deleting previously added taps. Using TapHandler involves adding taps via a call to an instance of the class; once all taps have been added, a call is made to finalize all scripts, i.e., write each script to file. When adding a tap, the caller passes the name of the tap via a parameter; recall from Section 3.3.1 and Table 3.1 that the name of the tap may be used to derive the name of the owning switch as well as the mac address of the corresponding net device, these relationships are used to correctly construe the content of each script. In the process of accepting taps to add, TapHandler also adds a corresponding instance of the class TapBridge in ns-3; recall from Figure 2.3 that TapBridge is the ns-3 class which interacts with and connects to TAP devices created on the operating system.
Listing 2 illustrates three example scripts that add two taps, add two corresponding ports, and finally remove said taps — in that order. The attentive reader may infer that the two taps belong to switches $s_1$ and $s_2$, and that each tap is the first port to be added to either switch; note also how the MAC address is derived from the tap name according to Table 3.1. Finalizing the scripts involves writing each of the three scripts to file, while also directly executing the script which creates the taps; note that adding ports need to be done by a separate script because ns-3 first has to boot up Mininet and allow it to setup all required OVS nodes. Lastly, TapHandler also provides a function for executing the delete script, should ns-3 wish to do so at the end of any emulation session. Note that a user may manually execute any script since they exist on the file system.

A class named NodeHandler tracks the desired Mininet topology using four instances of FileHandler; these files contain: (I) a list of switches; (II) a list of hosts; (III) a list of host-to-switch links; (IV) a list or ARP commands, with one command per host. NodeHandler is properly used by adding virtual OVS nodes to it, which are then added to the list of nodes. The owner may in turn add devices which belong to any previously added virtual node, these requests are then passed along to an instance of TapHandler. The owner must also specify any host which is to be in the topology, which are then added to a list of hosts; an ARP command is also added for each host. Recall from Section 3.3.2 that the IP address of a host may be derived from the host name. Links between hosts and switches are specified last, and are likewise added to a list. When the call is made to finalize the topology, each list is written to a file.

Recall from Figure 2.3 that a ghost node holds each TAP-attached net device: Some help is required in handling each set of ghost nodes that conceptually belong to the same OVS. Since a wireless link module is to be used for each net device, the position assigned to each ghost node determines the connectivity possible between pairs of net devices. Semantically, the position belongs to the OVS node; in practice, NodeHandler ensures that every ghost node attached to the same OVS also is provided the same position in the
Listing 3: Example of ns-3 output for Mininet to parse.

Listing 3 illustrates the file output corresponding to Listing 1. The top example contains a list OVS nodes, while the content below is a list of hosts. The third list is of host-switch pairs to be linked, with the bandwidth and latency, respectively — with 0 meaning unlimited bandwidth. The bottom content depicts a script which may be executed in each host, thereby setting up complete ARP-tables.

The class `MininetHandler` inherits from `NodeHandler`. These classes would, in truth, probably have been merged had the code been re-factored one more time; as it is, there are two classes where there should perhaps only be one. `MininetHandler` implements the singleton pattern, and so a single instance of it may be accessed from anywhere in the code in a static manner. Listing 1 illustrates the use of this class, which serves as the front-end for the entire solution. Beyond the functions that are inherited, this class also accepts commands to be executed at certain times during a simulation; these commands are written to a file, so that Mininet may read and parse, and then execute them at the appropriate time. Another list contains various parameters which Mininet requires to operate correctly, these being the PID of the ns-3 process, whether to run the previously mentioned ARP script or not, which controller to use, and which forwarding script to execute. Note that listing 1 also depicts the use of another class, `WifiNetwork`, through which devices are added to `MininetHandler`. `WifiNetwork` serves only as a convenience; manual setup of
each WiFi device would have been required had this class not been used in Listing 1.

4.3.1 Handling of system commands

One interesting realization which occurred during the implementation was that it is, in some ways, preferable to have the emulator write system commands to a file, and then executing the file as a script; as opposed to executing them one at a time, or via C/C++ function calls. This was not the original approach, but the implementation changed to conform to this paradigm rather early on. One advantage was that the script serves as a kind of record over all the system commands used, which can be somewhat helpful in debugging the system. Another advantage is that the user may rerun scripts containing the aforementioned system commands if the emulator for some reason fails the first time around. One obvious example is the convenience of being able to run the delete-script for all TAP devices, if the emulator happens to crash. Another example is the add-script which creates and connects all TAP devices to their corresponding OVS nodes.

4.3.2 ns-3 ACK/CTS issue

WiFi net devices acknowledge correctly received packets at the link layer. Since wireless links are much more likely to corrupt packets, it is beneficial to retransmit packets over individual wireless links, rather than waiting for the end-host to notice dropped or corrupt packets. The ns-3 WiFi net device will not ACK packets unless they are addressed to the device in question. As such, it is crucial that the forwarding rules rewrite the destination mac address to correspond to the next receiving net device. If this is not done, the receiving device will fail to ACK the packet, whereupon the sending device will retransmit the packet several times. The symptom of this is that the throughput will be limited to about 12.5% of the full capacity. This issue is particularly important to point out since the solution is outside of the scope of the system itself; ultimately, a controller such as OpenDaylight (ODL) will be installed to manage the system as a master of the OVS nodes. Whoever
manages this controller will have to be aware of this issue: When forwarding over ns-3 WiFi, always forward packets using the MAC address of the next receiving net device.
4.4 The Control Socket API

The socket API operates according to the following paradigm: Any parameter, in a set of single-value parameters, may be fetched or changed — or at least, a controlling entity may safely attempt to perform either action for every parameter in the aforementioned set. The term *single value* refers to the fact that the API does not deal in compound types, such as C structs. Values are currently passed to the API in plain text; the sender will have to have some idea of the type used in the source code underneath the API. The API constitutes an ns-3 module on its own, and is not a mere extension to the WiFi link module. Table 4.2 enumerates the files which make up this module, while Figure 4.4 contains the UML diagram for the API. The next sections will cover the socket API, starting at the socket and moving inwards towards the link module; this is followed by an example in Section 4.4.5 to illustrate the execution order of a call to the API; Section 4.4.6 details the extent of the parameters available via the API.
Figure 4.5: Overview of the entire system.

<table>
<thead>
<tr>
<th>File</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>ctrl-interface.h</td>
<td>Defines message constants</td>
</tr>
<tr>
<td>ctrl-interface.cc</td>
<td>Socket management and message parsing</td>
</tr>
<tr>
<td>interface-switch.h</td>
<td>Pure virtual class</td>
</tr>
<tr>
<td>interface-switch.cc</td>
<td>Casting functions</td>
</tr>
<tr>
<td>interface-wifi.h</td>
<td>Virtual Get/Set-functions suitable for WiFi</td>
</tr>
<tr>
<td>interface-wifi.cc</td>
<td>Populate array with pointers to Get/Set-functions</td>
</tr>
</tbody>
</table>

Table 4.2: All the constituent parts of the socket API module.
### Table 4.3: UNIX-domain socket address format for the socket API.

<table>
<thead>
<tr>
<th>Socket location</th>
<th>Address format</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Local API socket</td>
<td>&quot;/tmp/[MAC]-link&quot;</td>
<td>&quot;/tmp/00:00:00:02:05:00-link&quot;</td>
</tr>
<tr>
<td>Controller socket</td>
<td>&quot;/tmp/[MAC]-ctrl&quot;</td>
<td>&quot;/tmp/00:00:00:02:05:00-ctrl&quot;</td>
</tr>
</tbody>
</table>

### Table 4.4: The 2 byte header format for Get/Set messages.

<table>
<thead>
<tr>
<th>Message type</th>
<th>Header format (2 bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Get call</td>
<td>11TT TT00 VVVV VVVV</td>
</tr>
<tr>
<td>Set call</td>
<td>11TT TT01 VVVV VVVV</td>
</tr>
</tbody>
</table>

### 4.4.1 Message parsing

The class `CtrlInterface` in the API module implements UNIX-domain socket communication, and contains the logic required to parse messages. This layer of the API contains all the logic which is identical regardless of which underlying link module is used; Figure 4.4 illustrates that one might, for instance, want to attach the API to both the WiFi module and the wired module. The name of this class indicates that it is closest to the controller. For the controlling entity, the most important issue is the address of the API socket, as well as the expected address of the socket at the actual controller or OVS; the formats of these two addresses are given in Table 4.3. Note that both addresses are derived from the MAC-address of the net device to which the API is attached.

The class `CtrlInterface` also handles the required ns-3 scheduling; the current solution schedules regular socket checks via the ns-3 scheduling system at intervals of 10 ms. This scheduling system allows for function calls to be placed at certain points in time during the simulation. While this approach is rather simple, a more sophisticated solution would entail having a separate thread listen to the socket; this would avoid using the ns-3 scheduler, and would entail a faster response to any query over the API. Given that the current approach is to check the API every 10 milliseconds, a random response delay ranging from 0 to 10 ms is introduced for any call.
The general format of a message defines a two byte header and an optional tail, of variable length, containing a value; a get-call does not have a tail, while a set-call does; however, the message the API responds with always has a tail. The value in any message is to be interpreted as a string, and must be cast to whatever type the parameter is supposed to be of at the API. Table 4.4 depicts the format on a bit level; the field denoted by the letter T is reserved for the type of the message — which is currently always string, for which reason these bits have no effect. A single bit is used to determine if the call is a get or set call. The parameter that the call involves is denoted by a 8 bit value marked v; as will be seen, the 8-bit value is used as the literal index into an array of function pointers pointing to the appropriate get/set-pair. The tail is not depicted in the format, and the API will not look for a tail unless the message type is set. At the controller side, the tail will contain a value for any get-response that properly mapped onto a variable at the API; if the mapping fails a status message is returned instead. For a set-response, the tail always contains a status message, e.g., ACCEPTED.

Note that the four bits which specify the type would be required if the length of the message was unknown at the receiver side; if the length could not be assumed, it would also be difficult to use a string tail, since it could be of variable length. Always passing values in the form of strings offers consistency, as well as a kind of decoupling between the controller and the API; the API need not know the type which the controller desires for each parameter, and vice versa; nor do they need to agree on a common type for individual parameters. The alternative to this consistency is that one side must know the parameter type at the other side. On the other hand, the always-string approach requires that each message is cast twice — although each cast is more predictable, and may be carried out using C++ string-streams. Furthermore, passing numbers in the form of strings may also be ineffective in terms of message length; consider the number of bytes which would be required for an irrational number such as π. A value of type double would, however, always be of a particular size for any given operating system.
```cpp
class InterfaceWifi : public InterfaceSwitch {
public:
    InterfaceWifi();

    void AddItem (void (InterfaceWifi::*get)(), void (InterfaceWifi::*set)());
    void CallGet (u_int8_t key);
    void CallSet (u_int8_t key);

    /// ** Type ******************************************
    virtual void GetType () { ReturnValue ((string)"WIFI"); }
    virtual void SetType () { Block (); }

    /// ** Misc. MAC ****************************************
    virtual void GetMacAddr () { ReturnNull (); }
    virtual void SetMacAddr () { Block (); }
    virtual void GetBSSID () { ReturnNull (); }
    virtual void SetBSSID () { Block (); }
    virtual void GetSSID () { ReturnNull (); }
    virtual void SetSSID () { Block (); }
    virtual void GetMacTxBitCount () { ReturnNull (); }
    virtual void SetMacTxBitCount () { Block (); }
    virtual void GetMacRxBitCount () { ReturnNull (); }
    virtual void SetMacRxBitCount () { Block (); }

    /// ** Queue MAC ************************************
    virtual void GetQueueSize () { ReturnNull (); }
    virtual void SetQueueSize () { Block (); }
    virtual void GetQueueMaxSize () { ReturnNull (); }
    virtual void SetQueueMaxSize () { Block (); }
    virtual void GetQueueMaxDelay () { ReturnNull (); }
    virtual void SetQueueMaxDelay () { Block (); }
    virtual void GetDropBitCount () { ReturnNull (); }
    virtual void SetDropBitCount () { Block (); }
```

Listing 4: Multiplexing of WiFi API messages.
void
IntfWifi::AddItem ( void ( IntfWifi::*get)() , void ( IntfWifi::*set)() )
{
    m_intfSwitch.push_back (DemuxItem (get, set));
}

IntfWifi::IntfWifi ()
{
    AddItem (&IntfWifi::GetType,
             &IntfWifi::SetType);

    /** Misc. MAC **************************************************/
    AddItem (&IntfWifi::GetMacAddr,
             &IntfWifi::SetMacAddr);
    AddItem (&IntfWifi::GetBSSID,
             &IntfWifi::SetBSSID);
    AddItem (&IntfWifi::GetSSID,
             &IntfWifi::SetSSID);
    AddItem (&IntfWifi::GetMacTxBitCount,
             &IntfWifi::SetMacTxBitCount);
    AddItem (&IntfWifi::GetMacRxBitCount,
             &IntfWifi::SetMacRxBitCount);

    /** Queue MAC **************************************************/
    AddItem (&IntfWifi::GetQueueSize,
             &IntfWifi::SetQueueSize);
    AddItem (&IntfWifi::GetQueueMaxSize,
             &IntfWifi::SetQueueMaxSize);
    AddItem (&IntfWifi::GetQueueMaxDelay,
             &IntfWifi::SetQueueMaxDelay);
    AddItem (&IntfWifi::GetDropBitCount,
             &IntfWifi::SetDropBitCount);

    /** Misc. PHY **************************************************/
    AddItem (&IntfWifi::i_GetPhyState,
             &IntfWifi::i_SetPhyState);
    AddItem (&IntfWifi::i_GetRxAntennaCt,
             &IntfWifi::i_SetRxAntennaCt);
    AddItem (&IntfWifi::i_GetTxAntennaCt,
             &IntfWifi::i_SetTxAntennaCt);
    AddItem (&IntfWifi::i_GetIntfCount,
             &IntfWifi::i_SetIntfCount);
    AddItem (&IntfWifi::i_GetBandSupport,
             &IntfWifi::i_SetBandSupport);
}

Listing 5: Example content from the source code of InterfaceWifi.
4.4.2 Message casting

The second layer is implemented by InterfaceSwitch and all its subclasses; this section concerns InterfaceSwitch. The base class contains logic for casting values from C++ strings into other types. This is required since the outer layer handles strings only, and since most of variables in the actual link module is of other types.

One interesting aspect of this layer is the manner in which parameters are passed between the InterfaceSwitch-functions and the net device below. The goal for this class and the immediate subclasses is to be type agnostic, meaning it should not have to specify the type of the parameters passed between it and the net device. Consider a scenario in which the underlying net device has to pass a parameter up to InterfaceSwitch to complete a get-call: The net device calls a set of overloaded functions in InterfaceSwitch with the variable in question, where the set covers the desired type-space; this way, the C++ type system resolves the issue. Consider the opposite scenario where a variable is to be set in the net device: The call to set the variable will come from a subclass of InterfaceSwitch, but we wish that this class should not need to know the type. The call is therefore to notify the underlying net device that a value is available for the given parameter, the net device will then fetch this value and update the parameter. To handle this request InterfaceSwitch is equipped with several get-functions which return the value after casting it to the type which the net device requests. In this case the C++ type system does not resolve the issue, but the burden of knowing the type of each variable is moved from the API module to the link module; of course, the link module is much closer to this information. The fact that the socket API does not have knowledge of underlying types is visible in Listing 4; note how the constructor populates an array of function pointers. As an example, message 1 gets or sets the mac address, message 2 gets or sets the BSSID, message 2 gets or sets the SSID and so on. Listing 5 depicts the partial header of InterfaceWifi; note the numerous virtual Get/Set-functions, these will need to be overwritten by a subclass if a controller is to be able to communicate with the underlying
WiFi net device.

4.4.3 Message demultiplexing

The demultiplexing action occurs at the immediate subclasses of InterfaceSwitch; since this logic will differ from one link module to another, several different subclasses are required. Figure 4.4 indicates two subclasses, one for WiFi and one for the wired link module in ns-3. These classes receive messages from the outer layer and demultiplex these by consulting an *array of function pointer pairs*, the index into which is held in the message header. An example case from the WiFi demux is depicted in Listings 4 and 5; note the large number of virtual get/set-functions that are defined in the header in Listing 4, note also how these functions are added to the aforementioned array in Listing 5. Obviously, any entity that wishes to communicate with the API need to know the order in which these functions were added to the array. Thus we may say that this communication protocol entails two things: (I) the *header format* as defined in table 4.4; (II) the *mapping* between integers and variables as illustrated in Listing 7.

Because the code which is partially depicted in Listing 4 and 5 is so very repetitive, a simple python script was developed to auto-generate this code. This script takes as input a series of variable names and derives a list of variables; this list is then expanded so that there is instead two lists of function names, one to get and one to set each variable. The script then generates C++ code which constitutes the actual demux logic, located in a single class which inherits from InterfaceSwitch. This Python script makes it very easy, and relatively safe, to generate new demux units for different ns-3 link modules. The script also generates a mapping between integers and variables which is highly useful when developing the controlling entity — this mapping end up in a file with the same name as the demux unit, with the file extension .t. The python script, along with any demux unit generated by it as well as the mapping for the demux unit, is located in the folder where the rest of the module is located, this being at ns3/src/ctrl-interface/model/.

46
Listings 6 and 7 illustrate the use of the script and the mapping, respectively.

```python
def createWifiApi( ):
    wifiApi = ApiWriter( "wifi" )
    wifiApi.denoteCategory( "Misc. MAC" )
    wifiApi.addParam( "MacAddr" )
    wifiApi.addParam( "BSSID" )
    wifiApi.addParam( "SSID" )
    wifiApi.addParam( "MacTxBitCount" )
    wifiApi.addParam( "MacRxBitCount" )
    wifiApi.denoteCategory( "Queue MAC" )
    wifiApi.addParam( "QueueSize" )
    wifiApi.addParam( "QueueMaxSize" )
    wifiApi.addParam( "QueueMaxDelay" )
    wifiApi.addParam( "DropBitCount" )
    wifiApi.denoteCategory( "Misc. PHY" )
    wifiApi.addParam( "PhyState" )
    wifiApi.addParam( "RxAntennaCt" )
    wifiApi.addParam( "TxAntennaCt" )
```

Listing 6: Example of automated API generation.

4.4.4 Message-to-variable mapping

The classes described in previous sections in this chapter have all been part of the socket API module in ns-3, this section instead describes the modifications required at the net-device end — i.e., in the link module as opposed to the API module. One could make modifications to an existing class in the WiFi module, but that would entail that every single WiFi net device would have this API; the solution opted for in the case of WiFi is instead to create a new class which may be instantiated with any WiFi net device that required the API. Therefore, the class `InnerLinkInterface` in Figure 4.4 is added the WiFi module — whereas all the parent classes in that figure are in the socket API. The basic responsibility of this new class is to actually have access to all the various parameters that are visible via the socket API, and to map requests made via the API to these parameters,
Listing 7: Example of auto generated API mapping.

which are scattered throughout many different classes. The details of where each parameter is located is partially covered in Section 4.4.6.

4.4.5 Example call

The following example assumes that a WiFi net device is hooked up to the API described in this chapter via inner-link-interface.cc, which is visible in Figure 4.4. The example illustrates the execution flow when a call is made to get the channel number of this device. Any message is first received at an instance of ctrl-interface.cc; the first function in the execution is CheckSocket(), where the actual socket-read takes place. Any message received at this point is handled by ParseMessage() in the same class; in this example, the parsing process determines that a variable is to be fetched, and the index of that variable. In the case of a set-call, a parameter value would follow after the 2 byte header; however, in this case the call is to get the channel number. At this point, the process
enters `interface-wifi.cc` with a call to `CallGet(uint8_t key)`, where the parameter in the call serve as an index into an array of function pointer pairs; recall that this index was previously parsed from the socket message.

Where the execution goes next depends on whether `inner-link-interface.cc` overwrites the set function in `interface-wifi.cc`. The first case is that the function called is not overwritten, in which case the next call is to `GetChannelNr()` in `interface-wifi.cc`, which in turn call `ReturnNull()` in the parent class; this function stamps the index into a header and appends a null-answer. Next `SendReply(string msg)` is called to do the actual socket-write, thus completing the truncated get-call.

The second case is that `inner-link-interface.cc` indeed has overwritten the proper function in `interface-wifi.cc`, thus: `GetChannelNr()` in `inner-link-interface.cc` is called. This function contains the logic required to actually get the channel number from somewhere within the WiFi module. The actual channel number is thus fetched and `ReturnValue(<type> val)` is called — which version of this function is determined by the type system of C++. From this point on, the execution is the same as in the previous example. As this example illustrates, `interface-wifi.cc` need not be aware of the type on any underlying variable, it simply provides the required number of overloaded `ReturnValue(<type> val)`-functions via the parent class. Likewise in the case of a set-call, the value is not set from `interface-wifi.cc`, instead the underlying subclass will have to request the message to set, with the type that is used in the WiFi module.

### 4.4.6 Current extent of the API

Most of the discussion so far has been about the socket API in the abstract, this section will discuss the extent to which the current API has been integrated into the WiFi module; note that this refers to the to shim layer exemplified in Figure 4.5. This section documents the parameters which may be retrieved and altered at the link module via API, as well as the semantic meaning and domain of each parameter. The section also covers the extensions
added throughout the WiFi module and the shim layer, as well as processing done in that shim layer. Ideally, these changes should be limited to the shim layer, but as will be seen, some alterations were required elsewhere in the module.

<table>
<thead>
<tr>
<th>Token</th>
<th>Get</th>
<th>Set</th>
<th>Type</th>
<th>Units</th>
</tr>
</thead>
<tbody>
<tr>
<td>MAC_ADDR</td>
<td>●</td>
<td></td>
<td>string</td>
<td></td>
</tr>
<tr>
<td>BSSID</td>
<td>●</td>
<td></td>
<td>string</td>
<td></td>
</tr>
<tr>
<td>SSID</td>
<td>●</td>
<td></td>
<td>string</td>
<td></td>
</tr>
<tr>
<td>MAC_TX_BYTE_COUNT</td>
<td>●</td>
<td></td>
<td>uint32_t</td>
<td>Bytes</td>
</tr>
<tr>
<td>MAC_RX_BYTE_COUNT</td>
<td>●</td>
<td></td>
<td>uint32_t</td>
<td>Bytes</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Token</th>
<th>Get</th>
<th>Set</th>
<th>Type</th>
<th>Units</th>
</tr>
</thead>
<tbody>
<tr>
<td>STA_INDEX</td>
<td>●</td>
<td>●</td>
<td>uint32_t</td>
<td></td>
</tr>
<tr>
<td>STA_MAC_ADDR</td>
<td>●</td>
<td></td>
<td>string</td>
<td></td>
</tr>
<tr>
<td>STA_ERROR_RATE</td>
<td>●</td>
<td></td>
<td>double</td>
<td>Percent</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Token</th>
<th>Get</th>
<th>Set</th>
<th>Type</th>
<th>Units</th>
</tr>
</thead>
<tbody>
<tr>
<td>QUEUE_BYTES</td>
<td>●</td>
<td></td>
<td>uint32_t</td>
<td>Bytes</td>
</tr>
<tr>
<td>QUEUE_PACKETS</td>
<td>●</td>
<td></td>
<td>uint32_t</td>
<td>Packets</td>
</tr>
<tr>
<td>QUEUE_MAX_BYTES</td>
<td>●</td>
<td>●</td>
<td>uint32_t</td>
<td>Bytes</td>
</tr>
<tr>
<td>QUEUE_MAX_PACKETS</td>
<td>●</td>
<td>●</td>
<td>uint32_t</td>
<td>Packets</td>
</tr>
<tr>
<td>QUEUE_MAX_DELAY</td>
<td>●</td>
<td>●</td>
<td>uint</td>
<td>Milliseconds</td>
</tr>
<tr>
<td>QUEUE_DROP_BYTE_CT</td>
<td>●</td>
<td></td>
<td>uint32_t</td>
<td>Bytes</td>
</tr>
</tbody>
</table>

Table 4.5: Exposed MAC layer, station and queue parameters.
<table>
<thead>
<tr>
<th>Token</th>
<th>Get</th>
<th>Set</th>
<th>Type</th>
<th>Units</th>
</tr>
</thead>
<tbody>
<tr>
<td>PHY_STATE</td>
<td>•</td>
<td>•</td>
<td>string</td>
<td></td>
</tr>
<tr>
<td>INTF_COUNT</td>
<td>•</td>
<td></td>
<td>uint</td>
<td></td>
</tr>
<tr>
<td>DEVICE_SNR</td>
<td>•</td>
<td></td>
<td>double</td>
<td></td>
</tr>
<tr>
<td>PHY_MODE_CT</td>
<td>•</td>
<td></td>
<td>uint</td>
<td></td>
</tr>
<tr>
<td>PHY_MODE</td>
<td>•</td>
<td></td>
<td>uint16_t</td>
<td></td>
</tr>
<tr>
<td>PHY_MODE_STR</td>
<td>•</td>
<td></td>
<td>string</td>
<td></td>
</tr>
<tr>
<td>PHY_TX_BYTE_COUNT</td>
<td>•</td>
<td></td>
<td>uint32_t</td>
<td>Bytes</td>
</tr>
<tr>
<td>PHY_RX_BYTE_COUNT</td>
<td>•</td>
<td></td>
<td>uint32_t</td>
<td>Bytes</td>
</tr>
<tr>
<td>CHANNEL_FREQ</td>
<td>•</td>
<td></td>
<td>uint32_t</td>
<td>MHz</td>
</tr>
<tr>
<td>CHANNEL_NR</td>
<td>•</td>
<td>•</td>
<td>uint8_t</td>
<td></td>
</tr>
<tr>
<td>CHANNEL_WIDTH</td>
<td>•</td>
<td>•</td>
<td>uint32_t</td>
<td>MHz</td>
</tr>
<tr>
<td>NOISE_FLOOR</td>
<td>•</td>
<td></td>
<td>double</td>
<td>dBm</td>
</tr>
</tbody>
</table>

Table 4.6: Exposed physical layer and energy parameters.

Counters for the transmitted and received bytes in both the MAC and physical layers had to be added to the WiFi module in the corresponding classes, at the corresponding layers. These modifications include the variables that hold the counters, code to increase these when packets are received or transmitted, as well as get-functions so that these counters may be accessed. Station management is done by first setting the station index, after which interactions can take place with the station; this is currently limited to fetching the error rate and the MAC address. Table 4.5 enumerates all the exposed parameters which fall under the headings of general MAC-layer, station management, and MAC-layer queue. The most interesting parameters in this group are, perhaps, the MAC layer byte counters and the queue size. Examples of tracing the byte counters via the socket API may be seen in Figures 5.2, 5.3, 5.4, and 5.5.
Table 4.7: Exposed mobility parameters.

<table>
<thead>
<tr>
<th>Token</th>
<th>Get</th>
<th>Set</th>
<th>Type</th>
<th>Units</th>
</tr>
</thead>
<tbody>
<tr>
<td>POSITION</td>
<td>•</td>
<td>•</td>
<td>string</td>
<td>x y z</td>
</tr>
<tr>
<td>VELOCITY</td>
<td>•</td>
<td>•</td>
<td>string</td>
<td>x y z</td>
</tr>
</tbody>
</table>

Table 4.8: Exposed utilization parameters.

<table>
<thead>
<tr>
<th>Token</th>
<th>Get</th>
<th>Set</th>
<th>Type</th>
<th>Units</th>
</tr>
</thead>
<tbody>
<tr>
<td>IDLE_TIME</td>
<td>•</td>
<td></td>
<td>uint32_t</td>
<td>microseconds</td>
</tr>
<tr>
<td>CCA_BUSY_TIME</td>
<td>•</td>
<td></td>
<td>uint32_t</td>
<td>microseconds</td>
</tr>
<tr>
<td>BUSY_TIME</td>
<td>•</td>
<td></td>
<td>uint32_t</td>
<td>microseconds</td>
</tr>
<tr>
<td>RX_TIME</td>
<td>•</td>
<td></td>
<td>uint32_t</td>
<td>microseconds</td>
</tr>
<tr>
<td>RX_OK_TIME</td>
<td>•</td>
<td></td>
<td>uint32_t</td>
<td>microseconds</td>
</tr>
<tr>
<td>RX_ERROR_TIME</td>
<td>•</td>
<td></td>
<td>uint32_t</td>
<td>microseconds</td>
</tr>
<tr>
<td>TX_TIME</td>
<td>•</td>
<td></td>
<td>uint32_t</td>
<td>microseconds</td>
</tr>
<tr>
<td>SWITCHING_TIME</td>
<td>•</td>
<td></td>
<td>uint32_t</td>
<td>microseconds</td>
</tr>
<tr>
<td>SLEEP_TIME</td>
<td>•</td>
<td></td>
<td>uint32_t</td>
<td>microseconds</td>
</tr>
</tbody>
</table>

The energy parameters are located in the physical layer, except for the total power consumption which is drawn from an energy source which has to be attached to the physical layer. A variable and a get-function for the PHY-layer SNR had to be added; the same is true for the receive and transmit byte counters at the physical layer. The utilization statistics are gathered with the help of the class `WifiPhyListener` which `InnerLinkInterface` inherits; this enables the latter to be notified when there is a change in state of the physical layer. The time spent in each state is then tracked in terms of microseconds. Table 4.6 contains physical layer parameters as well as energy consumption data. Channel management via the socket API may be seen in Figure 5.2; examples of tracing the channel utilization can be seen in Figure 5.3; tracing of the power consumption may be seen in Figure 5.4.
5 Evaluation

This chapter presents results attained while validating the implementation of the emulator. Section 5.1 briefly validates and discusses the interaction between the emulator and the controller, and ability of the controller to detect the ns-3-based topology; Section 5.2 focuses on tracing parameters and doing some basic management via the socket API; all plots and all management decisions that are illustrated in Section 5.2 stems from interactions between a mock controller and the API. Recall that a last step of this thesis was supposed to be using the emulator to carry out real research, this last step had to be dropped for reasons of time and because of a fundamental limitation that turned out to exist when integrating ns-3 and any real time component — this issue is discussed in Section 5.2.

5.1 Controller network discovery

Figure 5.1 depicts an example topology, as seen by a real SDN controller — in this case the OpenDaylight controller. A similar example was given in Figure 2.5, the difference lies in where hosts are attached, whereas the mesh topology is identical. The underlying network was in either case emulated using the integrated system detailed in this report. Not only has the controller in this example detected all the switching elements, it has also correctly interpreted the connectivity between each switch, and installed forwarding rules such that hosts can reach one another. If the network was managed in the traditional manner, as described above, no single entity would have had the view depicted in Figure 5.1. One could envision that the nodes in Figure 5.1 correspond to a set of small cell base stations, as in Figure 1.1, and that one or more switches are connected to the macro cell base station; in which case this would be a good example of a wmSDN small cell backhaul. Note that ODL actually missed one link in Figure 2.5, while detecting it correctly in Figure 5.1 — this link being between switches 5 and 6.
5.2 Tracing via socket API

This section contains the following four examples: (I) tracing the throughput for parallel links as the channels are changed, so as to reduce interference and improve throughput; (II) tracing link utilization of single link as the modulation scheme is altered, while the throughput remains constant; (III) tracing the power consumption during consecutive packet bursts; (IV) tracing throughput at the MAC and physical layers and the Rx and Tx rates at both sides, with two links that approach one another.
One aspect of wireless communication which is of particular interest is the interference which a transmission may cause other transmissions, given that they are close enough to one another, both in space and in the frequency domain. Figure 5.2 illustrates a simple scenario in which there are four parallel flows, each running an Iperf session using UDP with a bit rate of 4 Mbps. Each of these flows are on the same WiFi channel, and will therefore interfere with one another. However, every 20 seconds one flow is moved to another channel; i.e., a pair of radios, one transmitting data while the other is receiving, switches channel simultaneously according to directives from the mock controller. After 3 such events, no interference should be observed, and all flows should get the desired 4 Mbps throughput. The experiment is carried out with a constant rate manager. The forwarding rules are static. All tracing and management is done via the socket API.
Figure 5.3: Channel utilization monitoring via controller.

Figure 5.3 illustrates the outcome when the channel utilization is monitored for a single WiFi link over which 0.5 Mbps of UDP traffic is forwarded using Iperf. The plot stacks the utilization of each radio as perceived through the socket API; the outcome should be that the time spent in different states sum up to 100 %. At the start of the emulation each radio is apparently set to have a modulation scheme yielding 1 Mbps, since 0.5 Mbps require a 50 % channel utilization. Every 10 seconds the controller asks each radio to change modulation scheme to the next higher rate; the outcome is a drop in the time spend transmitting and receiving, while the throughput remains constant. The reason for the outcome is, of course, that it requires less time to transmit 0.5 Mbps every time the modulation scheme is altered to increase the maximum rate. By the end the modulation scheme should allow for 11 Mbps, which would result in 4.5 % utilization.
Figure 5.4: Power consumption monitoring via controller.

Figure 5.4 illustrates the view of the mock controller as it monitors the power consumption of two radios on opposite sides of a link. Iperf is used to generate increasingly large bursts of UDP traffic starting shortly after the 25-second mark; with each increasing burst a greater power consumption is seen. Note that prior to the 20-second mark, the sender was in a sleeping mode, wherefore we see the exceedingly low power consumption for the sender at that time. Note also how the throughput becomes irregular as the rate approaches 5 Mbps during the last burst; the reason for this is the fact that the computer on which this was executed could barely process 5 Mbps in real-time. Of course, if this had been executed as a simulation, the simulation would have simply taken longer to finish. This is an unfortunate finding, as it turns out that this integrated emulator is limited to very low data rates. Recall Figure 5.2 where 16 Mbps was processed with seemingly no trouble; the reason for this is that this emulation was carried out on a better machine.
Lastly, Figure 5.5 plots the behavior when two WiFi links are created using the same channel; one link — both AP and station — then approach the other while the controller monitors the queue size, drop rate, queue limit, as well as byte rates on each radio; note that the rate is monitored on both the physical layer and a MAC layer. As before, Iperf is used to generate UDP traffic at a rate of 0.8 Mbps of each link. Looking at the lower plot, interference starts at around the 15 second mark, at which time the transmit rate plummets at the physical layer but not the MAC layer; as a result, the queue starts to increase. The reason why the upper and lower image in Figure 5.5 differ is because a different modulation scheme is used for either link; with the upper link using a 1 Mbps modulation, while the lower link uses an 11 Mbps modulation.
6 Discussion

This chapter will begin with an attempt to evaluate the outcome of the work described in this thesis in Section 6.1, where the consideration is both the quality of the work done and the feasibility of the emulator as it currently stand; Section 6.2 discusses future work; lastly, Section 6.3 offers some final words.

6.1 Evaluation

The first two goals were to design and implement a data plane integration followed by a control plane integration between OVS and ns-3. Both of these goals were achieved in reasonable time and with seemingly stable results, although not without some struggle. Ultimately, the system does what it is supposed to do in these regards. For a truly robust system, more testing of the socket API in particular would have been warranted.

With the data and control planes integrated, the idea was to integrate a newly made extension to the WiFi module which adds mmWave capabilities to the existing module. The first thing to note is that it was never clear if this addition would be released in time for it to be used in this thesis; as it turned out, the module became available towards the end the project. However, the module did not at that time include support for ad-hoc mmWave communication — though non-mmWave ad-hoc remained a possibility by default. This was a setback to the thesis as it would limit emulations to mmWave WiFi or adhoc WiFi, but not both.

With the aforementioned modules in place, the fourth goal was to validate the system, to make sure it worked as intended and that the system was in-fact useful. Another issue was revealed during this process; namely, the very low data rate which ns-3 is able to process in real time. This effectively limits any emulation topology to a very small size, and likewise limits any application to very low sending rate. One viable remaining use for the emulator is meshed sensor network, and robust forwarding over such networks.
The fifth and last goal was to use the emulator to look for interesting results in terms of robust routing and fast-failure. As it turned out, no time remained for serious research to be carried out using the emulator; however, this would have been difficult in any case because of the real-time constraint on ns-3.

6.2 Future Work

Work could be done to increase the effectiveness of ns-3. One straightforward approach is to boot several ns-3 processes, for instance one for every link or one for each net device. This could be a cheap way to turn ns-3 into a multi-core program — which it is not by default. Note, however, that this would limit the interaction between different links, since they are executed as different processes; thus there would, for instance, not be any interference between different pairs of radios. Another issue would be that the topology would become static if each radio pair existed in solitude in separate processes. It would also not be difficult to implement highly optimized lightweight wired link modules into ns-3, although the reason for using ns-3 was originally to gain access to sophisticated wireless modules.

One approach that could, perhaps, work is to run ns-3 in simulation time while still integrating it with OVS, but with an ns-3 controller; this might well work as long as there is no time component to the operation of OVS. Consider the following scenario: ns-3 operates in simulated time with a heavy traffic load while OVS operate in real-time, it naturally follows that ns-3 will perceive a long duration with very high data rates, while OVS perceive a short duration with very low data rates — there is nothing inherently disqualifying about this discrepancy in perception. Consider instead this: Fast-failover is used and therefore OVS generates BFD packets at a steady rate over some link, ns-3 will see a much 'thicker' BFD flow with the intervals between BFD packets appearing shorter than they do in real-time, thus also making the response time artificially good while also causing BFD-protocol packets to eat a larger proportion of the link capacity. In this case we see an issue because there is a time component to how fast-failover operates.
If altering ns-3 seems unattractive, the emulator is still useful for low data-rate emulations such as sensor network emulations. The module could be used to test robust routing over multihop meshed sensor networks; in fact, this is almost the only thing the emulator seems useful for right now.

6.3 Conclusion

In conclusion, the work itself has gone relatively well, with most of the things that was asked for having been implemented. The outcome itself is, however, somewhat disappointing. Some of the outcome could, perhaps, have been predicted. It is for instance not that surprising in hindsight that the real-time constraint turned out to be harsh on ns-3.
References


