

# The User Manual



Ramon dos Reis Fontes and Christian Esteve Rothenberg

INFORMATION & NETWORKING TECHNOLOGIES RESEARCH & INNOVATION GROUP (INTRIG)  
DEPARTMENT OF COMPUTER ENGINEERING AND INDUSTRIAL AUTOMATION (DCA)  
SCHOOL OF ELECTRICAL AND COMPUTER ENGINEERING (FEEC)  
UNIVERSITY OF CAMPINAS(UNICAMP) - BRAZIL

[GITHUB.COM/INTRIG-UNICAMP/MININET-WIFI](https://github.com/intrig-unicamp/mininet-wifi)

Mininet-WiFi is being developed as a clean extension of the high-fidelity Mininet emulator by adding the new abstractions and classes to support wireless NICs and emulated links while conserving all native lightweight virtualization and OpenFlow/SDN features.

*October 2019*





# Mininet-WiFi

Emulator for Software-Defined Wireless Networks

## Contents

<b>1</b>	<b>About Mininet-WiFi</b>	<b>9</b>
1.1	Requirements	9
1.2	Installing Mininet-WiFi	9
1.2.1	GitHub	9
1.2.2	Docker	10
1.3	Limitations	10
1.4	Architecture and Components	10
1.4.1	Classes	11
1.5	Wireless medium emulation	12
1.5.1	Traffic Control (TC)	12
1.5.2	Wmediumd	13
1.6	Usability	14
1.6.1	Changing Topology Size and Type	14
1.6.2	Examples	14
1.6.3	Getting information	14
1.6.4	Setting and getting node attributes via socket	15
1.6.5	Changes at runtime	15
1.7	Supported Features	16
1.7.1	Active Scanning	16
1.7.2	Bgscan (Background scanning)	16
1.7.3	Encryption	17
1.7.4	Hybrid Physical-Virtual Environment	17
1.7.5	IEEE 8021x	17
1.7.6	IEEE 802.11p	17

1.7.7	IEEE 802.11ax	20
1.7.8	IEEE 802.11r	20
1.7.9	Interference	20
1.7.10	Multiple interfaces	21
1.7.11	Multiple SSIDs over a single AP	21
1.7.12	Support to Virtual Interfaces	21
1.7.13	4-address for AP and client mode	22
1.7.14	Replaying Traces	22
<b>1.8</b>	<b>Client Isolation</b>	<b>23</b>
<b>1.9</b>	<b>Supported 802.11 Protocols</b>	<b>23</b>
<b>1.10</b>	<b>Videos</b>	<b>23</b>
<b>1.11</b>	<b>Contact Us</b>	<b>23</b>
<b>1.12</b>	<b>FAQ</b>	<b>23</b>
<b>2</b>	<b>Mobility Models</b>	<b>25</b>
<b>2.1</b>	<b>Mobility Models Supported</b>	<b>25</b>
<b>2.2</b>	<b>How to configure mobility models?</b>	<b>25</b>
<b>3</b>	<b>Propagation Models</b>	<b>27</b>
<b>3.1</b>	<b>Propagation Models Supported</b>	<b>27</b>
<b>3.2</b>	<b>How to add specific Propagation Model?</b>	<b>27</b>
<b>4</b>	<b>Tutorial</b>	<b>29</b>
<b>4.1</b>	<b>Introduction</b>	<b>29</b>
<b>4.2</b>	<b>First ideas</b>	<b>29</b>
4.2.1	How to read this post	29
4.2.2	Mininet-WiFi compared to Mininet	30
4.2.3	Mininet-WiFi and Mobility	30
4.2.4	802.11 Wireless LAN Emulation	31
4.2.5	Mininet-WiFi display graph	31
4.2.6	Install Mininet-WiFi on a Virtual Machine	31
4.2.7	Set up a new Ubuntu Server VM	31
4.2.8	Set up the Mininet-WiFi VM	32
<b>4.3</b>	<b>Mininet-WiFi Tutorial #1: One access point</b>	<b>32</b>
4.3.1	Capturing Wireless control traffic in Mininet-WiFi	32
4.3.2	Wireless Access Points and OpenFlow	33
4.3.3	Stop the tutorial	34
<b>4.4</b>	<b>Mininet-WiFi Tutorial #2: Multiple access points</b>	<b>35</b>
4.4.1	A simple mobility scenario	37
4.4.2	OpenFlow flows in a mobility scenario	37

4.4.3	Stop the tutorial	43
<b>4.5</b>	<b>Mininet-WiFi Tutorial #3: Python API and scripts</b>	<b>43</b>
4.5.1	The Mininet-WiFi Python API	43
4.5.2	Basic station and access point methods	43
4.5.3	Classic Mininet API	44
4.5.4	Example	45
4.5.5	Working at runtime	46
4.5.6	Mininet-WiFi CLI	46
4.5.7	Position	47
4.5.8	Distance	47
4.5.9	Mininet-WiFi Python runtime interpreter	47
4.5.10	Getting network information	47
4.5.11	Changing the network during runtime	48
4.5.12	Running commands in nodes	48
4.5.13	Mininet-WiFi and shell commands	49
4.5.14	Stop the tutorial	49
<b>4.6</b>	<b>Mininet-WiFi Tutorial #4: Mobility</b>	<b>49</b>
4.6.1	Python API and mobility	50
4.6.2	Moving a station in virtual space	50
4.6.3	Re-starting the scenario	52
4.6.4	More Python functions	52
4.6.5	Test with iperf	53
4.6.6	Stop the tutorial	53
<b>4.7</b>	<b>Mininet-WiFi Tutorial #5: VANETs (Veicular Ad Hoc Networks)</b>	<b>53</b>
4.7.1	Python API	53
4.7.2	Node Car Architecture	54
4.7.3	Integration with SUMO (Simulation of Urban Mobility)	55
<b>4.8</b>	<b>Mininet-WiFi example scripts</b>	<b>55</b>
<b>4.9</b>	<b>Conclusion</b>	<b>55</b>
<b>5</b>	<b>Guided exercises/demo</b>	<b>57</b>
<b>5.1</b>	<b>Guided exercises/demo</b>	<b>57</b>
5.1.1	Activity 1: Warming up	57
5.1.2	Activity 2: Loading Network Topologies	58
5.1.3	Activity 3: Customizing the Wireless Channel	59
<b>5.2</b>	<b>Further hands-on</b>	<b>59</b>
5.2.1	Activity 4: Mobility	59
5.2.2	Activity 5: Received Signal Strength	59
<b>5.3</b>	<b>OpenFlow</b>	<b>60</b>
5.3.1	Activity 6: Mobility and OpenFlow	60

<b>6</b>	<b>Reproducible Research</b>	<b>61</b>
<b>6.1</b>	<b>SwitchOn 2015</b>	<b>61</b>
<b>6.2</b>	<b>SBRC 2016</b>	<b>61</b>
6.2.1	Case 1 - Simple test	62
6.2.2	Case 2 (1/2) - Communication among stations and hosts/Verifying flow table	62
6.2.3	Case 2 (2/2) - Changing the controller (from reference to external controller)	62
6.2.4	Case 3 - Handover	62
6.2.5	Case 4 - Changes at runtime	62
6.2.6	Case 5 - Bridging physical and virtual emulated environments	63
<b>6.3</b>	<b>SIGCOMM 2016</b>	<b>63</b>
<b>6.4</b>	<b>From Theory to Experimental Evaluation: Resource Management in Software-Defined Vehicular Networks 2017</b>	<b>64</b>
<b>6.5</b>	<b>The Computer Journal - How far can we go? Towards Realistic Software-Defined Wireless Networking Experiments 2017</b>	<b>64</b>
6.5.1	<i>Wireless n-Casting</i>	64
6.5.2	<i>Multipath TCP</i>	65
6.5.3	<i>Hybrid Physical-Virtual Environment</i>	65
6.5.4	<i>SSID-based Flow Abstraction</i>	65
6.5.5	<i>EXPERIMENTAL VALIDATION: Propagation Model</i>	65
6.5.6	<i>EXPERIMENTAL VALIDATION: Simple File Transfer</i>	65
6.5.7	<i>EXPERIMENTAL VALIDATION: Replaying Network Conditions</i>	65
6.5.8	<i>On the Krack Attack: Reproducing Vulnerability and a Software-Defined Mitigation Approach WCNC 2018</i>	66
<b>7</b>	<b>Publications</b>	<b>69</b>
<b>7.1</b>	<b>SDN For Wireless 2015</b>	<b>69</b>
<b>7.2</b>	<b>CNSM 2015 - Best Paper Award!</b>	<b>69</b>
<b>7.3</b>	<b>SwitchOn 2015</b>	<b>69</b>
<b>7.4</b>	<b>SBRC 2016</b>	<b>69</b>
<b>7.5</b>	<b>SIGCOMM 2016</b>	<b>70</b>
<b>7.6</b>	<b>Institute of Electrical and Electronics Engineers - IEEE 2017</b>	<b>70</b>
<b>7.7</b>	<b>The Computer Journal 2017</b>	<b>70</b>
<b>7.8</b>	<b>WCNC 2018</b>	<b>70</b>
<b>8</b>	<b>Acknowledgment</b>	<b>71</b>
<b>9</b>	<b>Appendix</b>	<b>73</b>
<b>9.1</b>	<b>handover.py</b>	<b>73</b>





# Mininet-WiFi

Emulator for Software-Defined Wireless Networks

## 1. About Mininet-WiFi

Mininet-WiFi is a fork of the Mininet SDN network emulator and extended the functionality of Mininet by adding virtualized WiFi stations and access points based on the standard Linux wireless drivers and the 80211\_hwsim wireless simulation driver. We added classes to support the addition of these wireless devices in a Mininet network scenario and to emulate the attributes of a mobile station such as position and movement relative to the access points.

The Mininet-WiFi extended the base Mininet code by adding or modifying classes and scripts. So, Mininet-WiFi adds new functionality and still supports all the normal SDN emulation capabilities of the standard Mininet network emulator.

### 1.1 Requirements

Mininet-WiFi should work fine in any Ubuntu Distribution from 14.04.

### 1.2 Installing Mininet-WiFi

#### 1.2.1 GitHub

You have to follow only for 4 steps to install Mininet-WiFi:

- `sudo apt-get install git`
- `git clone https://github.com/intrig-unicamp/mininet-wifi`
- `cd mininet-wifi`
- `sudo util/install.sh -Wlnfv`

Mininet WiFi is installed by a script. Run the script with the `-h` `help` option to see all the options available.

```
wifi:~$ util/install.sh -h
```



Figure 1.1: Mininet-WiFi Components.

### 1.2.2 Docker

Mininet-WiFi is available on [Docker](#)<sup>1</sup>.

## 1.3 Limitations

Mininet-WiFi inherits all limitations of Mininet.

## 1.4 Architecture and Components

The main components that make part of the development of Mininet-WiFi are illustrated in Figure 1.1. In the kernel-space the module *mac80211\_hwsim* is responsible for creating virtual Wi-Fi interfaces, important for stations and access points. Continuing in the kernel-space, MLME (*Media Access Control Sublayer Management Entity*)<sup>2</sup> is realized in the stations side, while in the user-space the *hostapd* is responsible for this task in the AP side.

Mininet-WiFi also uses a couple utilities such as *iw*, *iwconfig* e o *wpa\_supplicant*. The first two are used for interface configuration and for getting information from wireless interfaces and the last one is used with *Hostapd*, in order to support WPA (*Wi-Fi Protected Access*), among other things. Besides them, another fundamental utility is *TC* (*Traffic Control*). The *TC* is an user-space utility program used to configure the Linux kernel packet scheduler, responsible for controlling the rate, delay, latency and loss, applying these attributes in virtual interfaces of stations and APs, representing with higher fidelity the behavior of the real world.

Figure 1.2 depicts the components and connections in a simple topology with two stations (or hosts) created with Mininet-WiFi, where the newly implemented components (highlighted in gray) are presented along the original Mininet building blocks. Although stations are equipped with a wireless interface by default they are able to connect with access points through wired links (veth pairs) as well.

More specifically, we added WiFi interfaces on stations that now are able to connect to an access point through its (*wlanX*) interface that is bridged to an OpenFlow switch with AP capabilities represented by (*ap1*). Similar to Mininet, the virtual network is created by placing host processes in

<sup>1</sup><https://registry.hub.docker.com/u/ramonfontes/mininet-wifi/>

<sup>2</sup>some of the functions performed by MLME are authentication, association, sending and receiving *beacons*, etc.



Figure 1.2: Components and connections in a two-host network created with Mininet-WiFi.

Linux OS network namespaces interconnected through virtual Ethernet (veth) pairs. The wireless interfaces to virtualize WiFi devices work on *master* mode for access points and *managed* mode for stations.

**Stations:** Are devices that connect to an access point through authentication and association. In our implementation, each station has one wireless card (*staX-wlan0* - where X shall be replaced by the number of each station). Since the traditional Mininet hosts are connected to an access point, stations are able to communicate with those hosts.

**Access Points:** Are devices that manage associated stations. Virtualized through *hostapd*<sup>3</sup> daemon and use virtual wireless interfaces for access point and authentication servers. While virtualized access points do not have (yet) APIs allowing users to configure several parameters in the same fashion of a real one, the current implementation covers the most important features, for example ssid, channel, mode, password, cryptography, etc.

Both stations and access points use *cfg80211* to communicate with the wireless device driver, a Linux 802.11 configuration API that provides communication between stations and *mac80211*. This framework in turn communicates directly with the WiFi device driver through a *netlink* socket (or more specifically *nl80211*) that is used to configure the *cfg80211* device and for kernel-user-space communication as well.

### 1.4.1 Classes

- *UserAP()*: User-space AP
- *OVSAP()*: Open vSwitch AP
- *addHost()*: adds a host to a topology and returns the host name
- *addAccessPoint()*: adds an access point to a topology and returns the access point name
- *addCar()*: adds a car to a topology and returns the car name
- *addStation()*: adds a station to a topology and returns the station name

<sup>3</sup>Hostapd (**H**ost **A**ccess **P**oint **D**aemon) user space software capable of turning normal wireless network interface cards into access points and authentication servers

- *addSwitch()*: adds a switch to a topology and returns the switch name
- *addLink()*: adds a bidirectional link to a topology (and returns a link key, but this is not important). Links in Mininet-WiFi are bidirectional unless noted otherwise.
- *plotGraph()*: use this class if you want to open the GUI.
- *startMobility*: useful when you want to use a mobility model.

## 1.5 Wireless medium emulation

Mininet-WiFi relies on two approaches for simulating the wireless medium: tc and wmediumd.

### 1.5.1 Traffic Control (TC)

Tc (traffic control) is the user-space utility program used to configure the Linux kernel packet scheduler. Used to configure Traffic Control in the Linux kernel, Traffic Control consists of the following:

- **Shaping**: When traffic is shaped, its rate of transmission is under control. Shaping may be more than lowering the available bandwidth - it is also used to smooth out bursts in traffic for better network behaviour. Shaping occurs on egress.
- **Scheduling**: By scheduling the transmission of packets it is possible to improve interactivity for traffic that needs it while still guaranteeing bandwidth to bulk transfers. Reordering is also called prioritizing, and happens only on egress.
- **Policing**: Where shaping deals with transmission of traffic, policing pertains to traffic arriving. Policing thus occurs on ingress.
- **Dropping**: Traffic exceeding a set bandwidth may also be dropped forthwith, both on ingress and on egress.

The aforementioned properties have been used to apply values for bandwidth, loss, latency and delay in Mininet-WiFi. Tc was the first approach adopted in Mininet-WiFi for simulating the wireless medium.

#### Intermediate Functional Block (IFB) Devices

There are two modes of traffic shaping: ingress and egress. Ingress handles incoming traffic and egress outgoing traffic. Linux does not support shaping/queuing on ingress, but only policing. Therefore IFB exists, which we can attach to the ingress queue while we can add any normal queuing like as egress queue on the IFB device.

Intermediate Functional Block (IFB) is an alternative to tc filters for handling ingress traffic, by redirecting it to a virtual interface and treat it as egress traffic. IFB is supported by setting `ifb=True` in `Mininet_wifi()`. Further information about IFB is available at [http://shorewall.net/traffic\\_shaping.htm#IFB](http://shorewall.net/traffic_shaping.htm#IFB).

#### Customizing Equations

When a node is in motion different values for bandwidth, latency, loss and delay are applied based on equations. Those equations can be customized by calling `setChannelEquation()`, for instance (it won't work for mesh/adhoc):

```
#dist = distance between receiver and transmitter
#custombw = is a default value for bw which takes into account the RSSI
net.setChannelEquation(bw='value.rate * (1.1 ** -dist)', loss='(dist * 2) / 100',
    ↪ delay='(dist / 10) + 1', latency='2 + dist')
```

### 1.5.2 Wmediumd

The kernel module `mac80211_hwsim` uses the same virtual medium for all wireless nodes. This means all nodes are internally in range of each other and they can be discovered in a wireless scan on the virtual interfaces. Mininet-WiFi simulates their position and wireless ranges by assigning stations to other stations or access points and revoking these wireless associations. If wireless interfaces should be isolated from each other (e.g. in adhoc or mesh networks) a solution like `wmediumd`<sup>4</sup> is required. It uses a kind of a dispatcher to permit or deny the transfer of packets from one interface to another.

A small helper to support `wmediumd` was initially developed by Patrick Große<sup>5</sup> and it was included in Mininet-WiFi. At the first release It supported two modes: *dynamic* and *static* mode.

#### Dynamic mode

This mode should be the best choice in most use cases. It requires the input of Mininet-WiFi interfaces as Python objects and retrieves the MAC addresses which are required for `wmediumd` automatically from the Mininet API.

#### Static mode

The static mode uses the MAC addresses that have been provided. In most cases the dynamic mode is the one to use. The advantage of the static mode is that no station objects are required and the data could be provided prior to the start of Mininet-WiFi.

Afterwards, support to interference was added. The support to interference improved the helper in such way that `wmediumd` is able to capture the position of nodes from Mininet-WiFi and it calculates the signal level based on the distance between source and destination by relying on the log distance propagation model. The interference model provided by `wmediumd` relies on the clear channel assessment (CCA) threshold, which defines a parameter causing interference.

*The initial idea behind both dynamic and static modes is now discarded since `wmediumd` can be used when you set `link=wmediumd`.*

#### Installation of wmediumd

It is required that `wmediumd` can be called using the command `wmediumd`. That means it should be located in `/usr/bin` or any other path that is available through the `PATH` environment variable. `wmediumd` can be automatically installed using the `-l` option on `util/install.sh`. This will copy the binary to `/usr/bin/wmediumd`.

**Installing wmediumd in Mininet-WiFi:** `sudo util/install.sh -l`

#### Traffic control versus Wmediumd

`Wmediumd` has been shown to be the best approach for the simulation of the wireless medium. Some advantages include:

- It isolates the wireless interfaces from each other
- `wmediumd` implements backoff algorithm. TC relies only in FIFO queue discipline.
- It decides when the association has to be evoked based on the signal level
- Values for bandwidth, loss, latency and delay are applied relying in a matrix. This matrix implements an option to determine PER (packet error rate) with outer matrix defined in

<sup>4</sup><https://github.com/bcopeland/wmediumd>

<sup>5</sup><https://github.com/patgrosse/wmediumd>

IEEE 802.11ax. The matrix is defined in Appendix 3 of *11-14-0571-12 TGax Evaluation Methodology*<sup>6</sup>.

- We highly recommend wmediumd for both adhoc and wireless mesh networks.

## 1.6 Usability

You can create a simple network (with 2 STAs and 1 AP) with the following command:

```
sudo mn --wifi
```

The command above can be used with other parameters, like in:

```
sudo mn --wifi --ssid=new_ssid --mode=g --channel=1
```

and also,

```
sudo mn --wifi --position --link=wmediumd --plot
```

*position* parameter automatically defines the position of the nodes, whereas *wmediumd* enables wmediumd (you can find more information about wmediumd along this manual). *plot* opens the GUI.

You can also use just mn if you want to work only with Mininet instead of Mininet-WiFi.

### 1.6.1 Changing Topology Size and Type

The default topology is a single Access Point connected to two Stations. You could change this to a different topo with `-topo` and pass parameters for that topology's creation. For example, to verify all-pairs ping connectivity with one Access Point and five Stations:

```
sudo mn --wifi --test pingall --topo single,5
```

Another example, with a linear topology (where each Access Point has one station, and all Access Points connect in a line via wired media):

```
sudo mn --wifi --test pingall --topo linear,5
```

### 1.6.2 Examples

If you are just beginning to write scripts for Mininet-WiFi, you can use the example scripts as a starting point. We created example scripts in the `/mininet-wifi/examples` directory that show how to use most of the features in Mininet-WiFi.

### 1.6.3 Getting information

**The commands below does not work with Wmediumd. In this case use wireless tools like iw/iwconfig instead.**

*getting the position:*

```
mininet-wifi>py sta1.params['position']
```

<sup>6</sup><https://mentor.ieee.org/802.11/dcn/14/11-14-0571-12-00ax-evaluation-methodology.docx>

*getting AP that a specific station is associated:*

```
mininet-wifi>py sta1.params['associatedTo']
```

*getting APs in range:*

```
mininet-wifi>py sta1.params['apsInRange']
```

*getting the channel:*

```
mininet-wifi>py sta1.params['channel']
```

*getting the frequency:*

```
mininet-wifi>py sta1.params['freq']
```

*getting the mode:*

```
mininet-wifi>py sta1.params['mode']
```

*getting the rssi:*

```
#not available when wmediumd and interference are enabled. Use iw/iwconfig  
↪ instead.  
mininet-wifi>py sta1.params['rssi']
```

*getting the Tx Power:*

```
mininet-wifi>py sta1.params['txpower']
```

*getting associated stations to ap1:*

```
mininet-wifi>py ap1.params['associatedStations']
```

#### 1.6.4 Setting and getting node attributes via socket

Please refer to examples/socket\_server.py and examples/socket\_client.py.  
Video demo: <https://www.youtube.com/watch?v=k69t9Xkb0nU>

#### 1.6.5 Changes at runtime

*setting the position (coord x, y, z):*

```
mininet-wifi>py sta1.setPosition('40,20,40')
```

*setting up the antenna gain (dBm):*

```
mininet-wifi>py sta1.setAntennaGain(5, intf='sta1-wlan0')
```

*setting the signal range (meters):*

```
mininet-wifi>py sta.setRange(100)  
mininet-wifi>py sta.setRange(100, intf='sta1-wlan0')
```

*setting Tx Power (dBm):*

```
mininet-wifi>py sta1.setTxPower(10, intf='sta1-wlan0')
```

*setting channel:*

```
mininet-wifi>py sta1.setChannel(10, intf='sta1-wlan0')
```

*changing association to ap1:*

```
mininet-wifi>py sta1.setAssociation(ap1, intf='sta1-wlan0')
```

*make interface work as adhoc mode:*

```
mininet-wifi>py sta1.setAdhocMode('sta1-wlan0', ssid='my-ssid')
```

*make interface work as mesh mode:*

```
mininet-wifi>py sta1.setMeshMode('sta1-wlan0', ssid='my-ssid')
```

## 1.7 Supported Features

### 1.7.1 Active Scanning

Space-separated list of frequencies in MHz to scan when searching for this BSS. If the subset of channels used by the network is known, this option can be used to optimize scanning to not occur on channels that the network does not use. Example: *scan\_freq="2412 2437 2462", freq\_list="2412 2437 2462"*

```
net.addStation("sta1", passwd="123456789a", encrypt="wpa2", active_scan=1,
↪ scan_freq="2412 2437 2462", freq_list="2412 2437 2462")
```

*freq\_list: List of frequencies*

*scan\_freq: List of frequencies to scan*

### 1.7.2 Bgscan (Background scanning)

wpa\_supplicant behavior for background scanning can be specified by configuring a bgscan module. This module is responsible for requesting background scans for the purpose of roaming within an ESS (i.e., within a single network block with all the APs using the same SSID). You can enable bgscan calling `setBgscan()`, for example:

```
net.setBgscan()
```

or

```
net.setBgscan(module="$module_name", s_interval=$value, signal=$value,
↪ l_interval=$value, database=$dir)
```

*s\_interval: short bgscan interval in seconds*

*signal: signal strength threshold*

*l\_interval: long interval*

*database: <database file name>*

source: [https://w1.fi/cgit/hostap/plain/wpa\\_supplicant/wpa\\_supplicant.conf](https://w1.fi/cgit/hostap/plain/wpa_supplicant/wpa_supplicant.conf)



### 1.7.3 Encryption

Mininet-WiFi supports all the common wireless security protocols, such as WEP (Wired Equivalent Privacy), WPA (Wi-Fi Protected Access) and WPA2.

On both station/ap side the password and encryption type can be set as below:

```
sta1 = net.addStation('sta1', passwd='123456789a', encrypt='wpa2')
ap1 = net.addAccessPoint('ap1', ssid="ssid1", mode="g", channel="1",
    ↪ passwd='123456789a', encrypt='wpa2')
```

**Attention! OVS does not support WPA/WPA2 authentication.**

### 1.7.4 Hybrid Physical-Virtual Environment

This feature was first presented in section 6.3 and features a hybrid physical-virtual environment where real users connect their 802.11-enabled smartphones to interact with the virtualized infrastructure, including nodes forming a mesh subnetwork, and access the global Internet after having its traffic processed through a multi-hop OpenFlow network.

### 1.7.5 IEEE 8021x

The example below enables 8021x.

```
ap1 = net.addAccessPoint( 'ap1', ... authmode='8021x', encrypt='wpa2', ...)
```

Further configuration includes *radius\_server* and *shared\_secret*:

```
ap1 = net.addAccessPoint( 'ap1', ... mode='8021x', encrypt='wpa2',
    ↪ enable_radius='yes', radius_server='127.0.0.1', shared_secret='secret'
    ↪ ...)
```

### 1.7.6 IEEE 802.11p

Please consider the procedures below if you want to enable with IEEE 802.11p.

#### wireless-regdb – regulatory information

Installing all the needed dependencies:

```
sudo apt-get install python-m2crypto
pip install future
```

Clone the repository, compile it, install it

```
wget https://mirrors.edge.kernel.org/pub/software/network/
    ↪ wireless-regdb/wireless-regdb-2018.05.31.tar.gz
tar -zxvf wireless-regdb-2018.05.31.tar.gz
cd wireless-regdb-2018.05.31
```

Your db.txt must include *country DE* as below:

```
country DE: DFS-ETSI
# entries 279004 and 280006
(2402 - 2482 @ 40), (20)
(5170 - 5250 @ 80), (20), AUTO-BW
(5250 - 5330 @ 80), (20), DFS, AUTO-BW
(5490 - 5710 @ 160), (27), DFS
# 5.9ghz band
# reference: http://www.etsi.org/deliver/etsi\_en/302500\_302599/302571/01.02.00\_20/en\_302571v010200a.pdf
(5850 - 5870 @ 10), (30)
(5860 - 5880 @ 10), (30)
(5870 - 5890 @ 10), (30)
(5880 - 5900 @ 10), (30)
(5890 - 5910 @ 10), (30)
(5900 - 5920 @ 10), (30)
(5910 - 5930 @ 10), (30)
# 60 GHz band channels 1-4, ref: Etsi En 302 567
(57240 - 65880 @ 2160), (40), NO-OUTDOOR

make maintainer-clean
make
sudo make install PREFIX=/
```

## CRDA – Central Regulatory Domain Agent

Install some extra packages

```
sudo apt-get install python-m2crypto libgrypt11-dev
```

Clone the repository

```
wget
↳ https://mirrors.edge.kernel.org/pub/software/network/crda/crda-3.18.tar.gz
tar -zxvf crda-3.18.tar.gz
cd crda-3.18
```

Copy your public key (installed by wireless-regdb, see above) to CRDA sources

```
cp /lib/crda/pubkeys/$USER.key.pub.pem pubkeys/
```

Compile and install CRDA (custom REG\_BIN is needed on Debian)

```
make REG_BIN=/lib/crda/regulatory.bin
sudo make install PREFIX=/ REG_BIN=/lib/crda/regulatory.bin
```

Test CRDA and the generated regulatory.bin

```
sudo /sbin/regdbdump /lib/crda/regulatory.bin | grep -i ocb
country 00: invalid
(5850.000 - 5925.000 @ 20.000), (20.00), NO-CCK, OCB-ONLY
```

**mac80211\_hwsim.c**

First you have to create a *Makefile* with the content below:

```
obj-m += mac80211_hwsim.o

all:
    make -C /lib/modules/$(shell uname -r)/build M=$(PWD) modules

clean:
    make -C /lib/modules/$(shell uname -r)/build M=$(PWD) clean
```

Then, you should consider mac80211\_hwsim with the changes below (consider to use the mac80211\_hwsim version that matches with your kernel version):

[https://github.com/torvalds/linux/blob/master/drivers/net/wireless/mac80211\\_hwsim.c](https://github.com/torvalds/linux/blob/master/drivers/net/wireless/mac80211_hwsim.c)

```
static const struct ieee80211_regdomain hwsim_world_regdom_custom_01
+ .n_reg_rules = 5,
+ REG_RULE(5855-10, 5925+10, 40, 0, 33, 0),

static const struct ieee80211_regdomain hwsim_world_regdom_custom_02 = {
+ .n_reg_rules = 3,
+ REG_RULE(5855-10, 5925+10, 40, 0, 33, 0),

static const struct ieee80211_iface_combination hwsim_if_comb[] = {
+.radar_detect_widths = BIT(NL80211_CHAN_WIDTH_5) |
                        BIT(NL80211_CHAN_WIDTH_10) |

static const struct ieee80211_iface_combination hwsim_if_comb_p2p_dev[] = {
+.radar_detect_widths = BIT(NL80211_CHAN_WIDTH_5) |
                        BIT(NL80211_CHAN_WIDTH_10) |

static const struct ieee80211_channel hwsim_channels_5ghz[]

/* ITS-G5B */
+ CHAN5G(5855), /* Channel 171 */
+ CHAN5G(5860), /* Channel 172 */
+ CHAN5G(5865), /* Channel 173 */
+ CHAN5G(5870), /* Channel 174 */
+ /* ITS-G5A */
+ CHAN5G(5875), /* Channel 175 */
+ CHAN5G(5880), /* Channel 176 */
+ CHAN5G(5885), /* Channel 177 */
+ CHAN5G(5890), /* Channel 178 */
+ CHAN5G(5895), /* Channel 179 */
+ CHAN5G(5900), /* Channel 180 */
+ CHAN5G(5905), /* Channel 181 */
+ /* ITS-G5D */
+ CHAN5G(5910), /* Channel 182 */
+ CHAN5G(5915), /* Channel 183 */
+ CHAN5G(5920), /* Channel 184 */
+ CHAN5G(5925), /* Channel 185 */
```

```

#ifdef CONFIG_MAC80211_MESH
                                BIT(NL80211_IFTYPE_MESH_POINT) |
#endif
                                BIT(NL80211_IFTYPE_AP) |
                                BIT(NL80211_IFTYPE_OCB) |
                                + BIT(NL80211_IFTYPE_P2P_GO) },

if (vif->type != NL80211_IFTYPE_AP &&
    vif->type != NL80211_IFTYPE_MESH_POINT &&
    + vif->type != NL80211_IFTYPE_OCB &&
    vif->type != NL80211_IFTYPE_ADHOC)
    return;

hw->wiphy->interface_modes = BIT(NL80211_IFTYPE_STATION) |
                                BIT(NL80211_IFTYPE_AP) |
                                BIT(NL80211_IFTYPE_P2P_CLIENT) |
                                BIT(NL80211_IFTYPE_P2P_GO) |
                                BIT(NL80211_IFTYPE_ADHOC) |
                                + BIT(NL80211_IFTYPE_OCB) |
                                BIT(NL80211_IFTYPE_MESH_POINT);

```

Finally, you run the *make* command to compile `mac80211_hwsim`.

### 1.7.7 IEEE 802.11ax

The example below enables 802.11ax.

```
ap1 = net.addAccessPoint( 'ap1', ... mode='ax' )
```

### 1.7.8 IEEE 802.11r

The example below enables 802.11r.

```
ap1 = net.addAccessPoint( 'ap1', ... passwd='123456789a', encrypt='wpa2',
    ↪ ieee80211r='yes', mobility_domain='a1b2' )
```

### 1.7.9 Interference

Mininet-WiFi supports the interference model provided by `wmediumd`. An example about how to enable interference is given below.

```
net = Mininet( ... link=wmediumd, wmediumd_mode=interference )
```

The interference model implemented in `wmediumd` relies on CCA THRESHOLD (Clear Channel Assessment). The CCA is used by the MAC layer to determine (i) if the channel is clear for transmitting data, and (ii) for determining when there is incoming data. Evaluation of CCA is made by the PHY layer and the resulting assessment is communicated to the MAC layer via the PHY-CCA.indicate service primitive. This primitive can either be set to IDLE, when the channel is assessed to be clear, or BUSY when the channel is assessed to be in use.

Optionally, a fading coefficient is also supported by Mininet-WiFi when `wmediumd` is enabled:

```
net = Mininet( ... link=wmediumd, wmediumd_mode=interference,
↳ fading_coefficient=1 )
```

Further details about the interference model implemented in `wmediumd` can be found at the master thesis titled *Medium and mobility behaviour insertion for 802.11 emulated networks*.

### 1.7.10 Multiple interfaces

Any wireless device can have multiple wireless interfaces. The parameter `wlans` allows you to add many interfaces on a single device. For example, let's take the code below:

```
sta1 = net.addStation( 'sta1', wlans=2)
```

`wlans=2` means that two wireless interfaces will be added for `sta1`. APs can have multiple wireless interfaces as well, however, they deserve a particular attention. For example, let's take the code below:

```
ap1 = net.addAccessPoint( 'ap1', wlans=2, ssid="ssid1,ssid2", mode="g",
↳ channel="1" )
```

You have to define two SSIDs separated by comma. If you do not want two SSIDs for some reason, you can do like below:

```
ap1 = net.addAccessPoint( 'ap1', wlans=2, ssid="ssid1,", mode="g", channel="1"
↳ )
```

Thus, the last interface will not have SSID.

### 1.7.11 Multiple SSIDs over a single AP

It is very common for an organization to have multiple SSIDs in their wireless network for various purposes, including: (i) to provide different security mechanisms such as WPA2-Enterprise for your employees and an “open” network with a captive portal for guests; (ii) to split bandwidth among different types of service; or (iii) to reduce costs by reducing the amount of physical access points. In Mininet-WiFi, an unique AP supports up to 8 different SSIDs (limitation imposed by `mac80211_hwsim`). Multiple SSIDs can be configured as below:

```
ap1 = net.addAccessPoint( 'ap1', ssid="ssid1,ssid2,ssid3", mode="g",
↳ channel="1" )
```

### 1.7.12 Support to Virtual Interfaces

The `mac80211` subsystem in the linux kernel supports multiple wireless interfaces to be created with one physical wireless card. To do so you have to set the number of virtual interfaces to be created (`nvif`), as illustrated below.

```
sta1 = net.addStation( 'sta1', nvif=2 )
```

### 1.7.13 4-address for AP and client mode

IEEE 802.11 (WLAN) frames have four address fields in their headers. In order to transport ethernet packets transparently over a Wireless Distribution System (WDS) link, the IEEE 802.3 (Ethernet) frame gets encapsulated in a IEEE 802.11 (WLAN) frame. In this case all of the four address fields are used, for:

- sender of the ethernet frame
- receiver of the ethernet frame
- sender of the WLAN frame
- receiver of the WLAN frame

Sender and receiver of the ethernet frame are simply copied from the transported ethernet frame. The remaining fields allow the receiver to recognize that the frame is meant for him, and allow it to acknowledge the reception of the frame to the (WLAN) sender. However, usually only three of the four fields are needed, so most drivers don't know about how to handle frames which make use of all four address fields. In other words: the most important ingredient for WDS is support for 4-address-headers.

Worth to mention that the 4-address frame format does not itself constitute a wireless distribution system or a wireless DS, it is simply a frame addressing mechanism that allows creation of a multitude of specialized implementations, one of which could be a wireless distribution system. The advantage of this mode compared to regular WDS mode is that it's easier to configure and does not require a static list of peer MAC addresses on any side.

A sample file for 4-address is available at */examples/4address.py*. In general, the user must set AP/Client with the param `_4addr`, for example:

```
ap1 = net.addAccessPoint( 'ap1', _4addr="ap", ssid="ssid1" ... )
ap2 = net.addAccessPoint( 'ap2', _4addr="client", ssid="ssid2" ... )
```

and then a 4-address link must be created as below:

```
net.addLink(ap1, ap2, link=_4addr)
```

### 1.7.14 Replaying Traces

Being able to replay real networking conditions based on traffic observations in real environments is useful to predict network performance under certain conditions, reason about the observed network behavior, and perform fair comparisons between alternative algorithms' implementations subject to the mirrors of the physical network. Mininet-WiFi is able to replay network conditions, bandwidth, mobility and rssi.

#### Replaying Network Conditions

Please refer to *examples/replaying/replayingNetworkConditions.py*

#### Replaying Bandwidth

Please refer to *examples/replaying/replayingBandwidth.py*

#### Replaying Mobility

Please refer to *examples/replaying/replayingMobility.py*

#### Replaying RSSI

Please refer to *examples/replaying/replayingRSSI.py*

## 1.8 Client Isolation

By default, stations associated with the same access point can communicate with each other without any OpenFlow rules. If you want to enable OpenFlow in such case, you need to use the client isolation. You can either try `sudo mn -wifi -no-bridge` or take `simplewifitopology.py` as reference.

## 1.9 Supported 802.11 Protocols

Mininet-WiFi already supports: IEEE 802.11a, IEEE 802.11b, IEEE 802.11g, IEEE 802.11h, IEEE 802.11i, IEEE 802.11n, IEEE 802.11q, IEEE 802.11s, IEEE 802.11r, IEEE 802.11x, IEEE 802.11ac, IEEE 802.11ax.

**Info:** `mac80211_hwsim` can be extended in order to support all protocols supported by `mac80211`!

## 1.10 Videos

You can find many videos about Mininet-WiFi in a [channel on youtube](#)<sup>7</sup>.

## 1.11 Contact Us

You are invited to participate of our [mailing list](#)<sup>8</sup>.

## 1.12 FAQ



How to solve the error: **IndexError: list index out of range**

```
sudo mn -c
```



Is it possible to create a wired link between station and ap?

Yes. When you add a link between station and ap you have to add the parameter `link='wired'` (e.g. `net.addLink(sta1, ap2, link='wired')`)



May I customize the Access Point and Station configuration file?

Since the Access Point is based on Hostapd you may customize the configuration file generated by Mininet-WiFi by adding the parameter `config` when you create the access point, for example:

```
ap1 = net.addAccessPoint( 'ap1',
    → config='ctrl_interface=/var/run/hostapd/,ctrl_interface_group=0' )
```

The code above creates a temp file called `ap1-wlan1.apconf`.

The same can be done for stations when `wpa_supplicant` is used, for example:

<sup>7</sup>[https://www.youtube.com/playlist?list=PLccoFREVAAt\\_4nEtrkl59mjff5ZzRX8DZA](https://www.youtube.com/playlist?list=PLccoFREVAAt_4nEtrkl59mjff5ZzRX8DZA)

<sup>8</sup><https://groups.google.com/forum/#!forum/mininet-wifi-discuss>

```
sta1 = net.addStation( 'sta1', config='eap=PEAP,identity="bob"' )
```



The mode N supports booth 2.4 and 5Ghz. How can I make a choice between 2.4 or 5Ghz?

You have to set band=2.4 or band=5 when you add an AP.



How to uninstall Mininet-WiFi?

```
sudo rm -rf /usr/local/bin/mn /usr/local/bin/mnexec /usr/local/lib/python*/*/mininet* /usr/local/bin/ovs-
* /usr/local/sbin/ovs-*
```



I do not have mac80211\_hwsim. How can I install it?

:

```
sudo apt-get install linux-image-extra-`uname -r`
```



How can I plot hosts and switches?

Syntax: node\_name.plot(position=x,y,z)

```
s1.plot(position='10,10,0')
```

However, some minor change in the Mininet base code is required. To do so, edit mininet/node.py by adding the code below into the Node() class:

```
def plot(self, position):
    self.params['position'] = position.split(',')
    self.params['range'] = [0]
    self.plotted = True
```

and then compile the code with `sudo make install`.





# Mininet-WiFi

Emulator for Software-Defined Wireless Networks

## 2. Mobility Models

### 2.1 Mobility Models Supported

Mininet-WiFi supports the following mobility models: RandomWalk, TruncatedLevyWalk, RandomDirection, RandomWayPoint, GaussMarkov, ReferencePoint and TimeVariantCommunity.

### 2.2 How to configure mobility models?

The mobility models can be configured by means of the method `net.startMobility()`, like in *examples/mobilityModel.py*. You may also have to consider the use of *examples/mobility.py* if you want a different type of mobility. In addition, you may want to taking into account some parameters:

#### RandomWalk

```
sta1 = net.addStation( ..., min_x=10, max_x=20, min_y=10, max_y=20,  
    ↪ constantDistance=1, constantVelocity=1 )
```

```
min_x = Minimum X position # default=0  
max_x = Maximum X position # default=100  
min_y = Minimum Y position # default=0  
max_y = Maximum Y position # default=100  
constantDistance = the value for the constant distance traveled in each step.  
    ↪ Default is 1.0.  
constantVelocity = the value for the constant node velocity. Default is 1.0
```

In the current implementation each step is 1 second. It is not possible to have a velocity larger than the distance.

#### RandomDirection

```
sta1 = net.addStation( ..., min_x=10, max_x=20, min_y=10, max_y=20, min_v=1,  
    ↪ max_v=2 )
```

```

min_x = Minimum X position # default=0
max_x = Maximum X position # default=100
min_y = Minimum Y position # default=0
max_y = Maximum Y position # default=100
min_v = Minimum value for node velocity
max_v = Maximum value for node velocity

```

### RandomWayPoint

```

sta1 = net.addStation( ..., min_x=10, max_x=20, min_y=10, max_y=20, min_v=1,
    ↪ max_v=2, min_wt=0, max_wt=100 )
//min_v and max_v must have different values

```

```

min_x = Minimum X position # default=0
max_x = Maximum X position # default=100
min_y = Minimum Y position # default=0
max_y = Maximum Y position # default=100
min_v = Minimum value for node velocity # default=5
max_v = Maximum value for node velocity # default=5
min_wt = Minimum wait time # default=0
max_wt = Maximum wait time # default=100

```

### GaussMarkov

```

sta1 = net.addStation( ..., min_x=10, max_x=20, min_y=10, max_y=20 )

min_x = Minimum X position # default=0
max_x = Maximum X position # default=100
min_y = Minimum Y position # default=0
max_y = Maximum Y position # default=100

```



# Mininet-WiFi

Emulator for Software-Defined Wireless Networks

## 3. Propagation Models

### 3.1 Propagation Models Supported

Mininet-WiFi supports the following propagation models: Friis Propagation Loss Model, Log-Distance Propagation Loss Model (DEFAULT ONE), Log-Normal Shadowing Propagation Loss Model, International Telecommunication Union (ITU) Propagation Loss Model and Two-Ray Ground Propagation Loss Model.

### 3.2 How to add specific Propagation Model?

You have to call the method `net.setPropagationModel()` like in *examples/propagationModel.py*. You might have to consider some parameters for specific propagation models (not mandatory), for example:

#### Friis Propagation Loss Model

```
net.setPropagationModel(model="friis", sL = $int)
sL = system loss
```

#### Log-Distance Propagation Loss Model

```
net.setPropagationModel(model="logDistance", sL = $int, exp = $int)
sL = system loss
exp = exponent
```

#### Log-Normal Shadowing Propagation Loss Model

```
net.setPropagationModel(model="logNormalShadowing", sL = $int, exp = $int,
    ↪ variance = $int)
sL = system loss
exp = exponent
variance = gaussian variance
```

**International Telecommunication Union (ITU) Propagation Loss Model**

```
net.setPropagationModel(model="ITU", lF = $int, nFloors = $int, pL = $int)
```

`lF` = floor penetration loss factor

`nFloors` = number of floors

`pL` = power loss coefficient

**Two-Ray Ground Propagation Loss Model**

*Attention: It does not give a good result for a short distance*

```
net.setPropagationModel(model="twoRayGround")
```



# Mininet-WiFi

Emulator for Software-Defined Wireless Networks

## 4. Tutorial

### 4.1 Introduction

This tutorial has been developed by Brian Linkletter (<http://www.brianlinkletter.com/mininet-wifi-software-defined-network-emulator-supports-wifi-networks>). We thank Brian for his time and this helpful tutorial.

### 4.2 First ideas

In this post, I describe the unique functions available in the Mininet-WiFi network emulator and work through a few tutorials exploring its features.

#### 4.2.1 How to read this post

In this post, I present the basic functionality of Mininet-WiFi by working through a series of tutorials, each of which works through Mininet-WiFi features, while building on the knowledge presented in the previous tutorial. I suggest new users work through each tutorial in order.

I do not attempt to cover every feature in Mininet-WiFi. Once you work through the tutorials in this post, you will be well equipped to discover all the features in Mininet-WiFi by working through the `Mininet-WiFi example scripts`- <https://github.com/intrig-unicamp/mininet-wifi/tree/master/examples>, and reading the `Mininet-WiFi wiki`<sup>1</sup> and `mailing list`<sup>2</sup>.

I assume the reader is already familiar with the [Mininet network emulator](<http://mininet.org/>) so I cover only the new WiFi features added by Mininet-WiFi. If you are not familiar with Mininet, please read my `Mininet network simulator review`<sup>3</sup> before proceeding. I have also written many other posts about Mininet<sup>4</sup>.

---

<sup>1</sup><https://github.com/intrig-unicamp/mininet-wifi/wiki>

<sup>2</sup><https://groups.google.com/forum/#!forum/mininet-wifi-discuss>

<sup>3</sup><http://www.brianlinkletter.com/mininet-test-drive/>

<sup>4</sup><http://www.brianlinkletter.com/tag/mininet/>

I start by discussing the functionality that Mininet-WiFi adds to Mininet: Mobility functions and WiFi interfaces. Then I show how to install Mininet-WiFi and work through the tutorials listed below:

Tutorial #1: One access point shows how to run the simplest Mininet-WiFi scenario, shows how to capture wireless traffic in a Mininet-Wifi network, and discusses the issues with OpenFlow and wireless LANs.

Tutorial #2: Multiple access points shows how to create a more complex network topology so we can experiment with a very basic mobility scenario. It discusses more about OpenFlow and shows how the Mininet reference controller works in Mininet-WiFi.

Tutorial #3: Python API and scripts shows how to create more complex network topologies using the Mininet-WiFi Python API to define node positions in space and other node attributes. It also discusses how to interact with nodes running in a scenario with the Mininet-WiFi CLI, the Mininet-WiFi Python interpreter, and by running commands in a node's shell.

Tutorial #4: Mobility shows how to create a network mobility scenario in which stations move through space and may move in and out of range of access points. It also discusses the available functions that may be used to implement different mobility models using the Mininet-WiFi Python API.

## 4.2.2 Mininet-WiFi compared to Mininet

Mininet-WiFi is an extension of the Mininet software defined network emulator. The Mininet-WiFi developer did not modify any existing Mininet functionality, but added new functionality.

## 4.2.3 Mininet-WiFi and Mobility

Broadly defined, mobility in the context of data networking refers to the ability of a network to accommodate hosts moving from one part of the network to another. For example: a cell phone user may switch to a wifi access point when she walks into a coffee shop; or a laptop user may walk from her office in one part of a building to a meeting room in another part of the building and still being able to connect to the network via the nearest WiFi access point.

While the standard Mininet network emulator may be used to test mobility (In the Mininet examples folder, we find a `mobility.py` script that demonstrates methods that may be used to create a scenario where a host connected to one switch moves its connection to another switch), Mininet-WiFi offers more options to emulate complex scenarios where many hosts will be changing the switches to which they are connected. Mininet-WiFi adds new classes that simplify the programming work required by researchers to create Mobility scenarios.

Mininet-WiFi does not modify the reference SDN controller provided by standard Mininet so the reference controller cannot manage the mobility of users in the wireless network. Researchers must use a remote controller that supports the CAPWAP protocol (NOTE: I've not tried this and I do not know if it will work without modifications or additional programming), or manually add and delete flows in the access points and switches.

#### 4.2.4 802.11 Wireless LAN Emulation

Mininet-wifi incorporates the Linux 802.11 SoftMAC<sup>5</sup> wireless drivers, the `cfg80211`<sup>6</sup> wireless configuration interface and the `mac80211_hwsim`<sup>7</sup> wireless simulation drivers in its access points.

The `mac80211_hwsim` driver is a software simulator for Wi-Fi radios. It can be used to create virtual wi-fi interfaces<sup>8</sup> that use the 802.11 SoftMAC wireless LAN driver<sup>9</sup>. Using this tool, researchers may emulate a Wi-Fi link between virtual machines<sup>10</sup> - some `mac80211_hwsim` practical examples and supporting information are at the following links: `lab`<sup>11</sup>, `thesis`<sup>12</sup>, `hostapd`<sup>13</sup>, `wpa-supPLICANT`<sup>14</sup>, `docs-1`<sup>15</sup>, and `docs-2`<sup>16</sup>. The `80211_hwsim` driver enables researchers to emulate the wifi protocol control messages passing between virtual wireless access points and virtual mobile stations in a network emulation scenario. By default, `80211_hwsim` simulates perfect conditions, which means there is no packet loss or corruption.

#### 4.2.5 Mininet-WiFi display graph

Since locations of nodes in space is an important aspect of WiFi networks, Mininet WiFi provides a graphical display (figure 4.1) showing locations of WiFi nodes in a graph. The graph may be created by calling its method in the Mininet-WiFi Python API (see examples in the tutorials below).

The graph will show wireless access points and stations, their positions in space and will display the affects of the range parameter for each node. The graph will not show any "wired" network elements such as standard Mininet hosts or switches, Ethernet connections between access points, hosts, or switches.

#### 4.2.6 Install Mininet-WiFi on a Virtual Machine

First, we need to create a virtual machine that will run the Mininet-WiFi network emulator. In the example below, we will use the VirtualBox virtual machine manager because it is open-source and runs on Windows, Mac OS, and Linux.

#### 4.2.7 Set up a new Ubuntu Server VM

Install Ubuntu Server in a new VM. Download an Ubuntu Server ISO image from the Ubuntu web site. See my post about installing Debian Linux in a VM<sup>17</sup>. Follow the same steps to install Ubuntu.

In this example, we will name the VM Mininet-WiFi.

<sup>5</sup><https://wireless.wiki.kernel.org/en/developers/documentation/glossary#softmac>

<sup>6</sup><http://www.linuxwireless.org/en/developers/Documentation/cfg80211>

<sup>7</sup>[https://wireless.wiki.kernel.org/en/users/drivers/mac80211\\_hwsim](https://wireless.wiki.kernel.org/en/users/drivers/mac80211_hwsim)

<sup>8</sup><http://stackoverflow.com/questions/33091895/virtual-wifi-802-11-interface-similar-to-veth-on-linux>

<sup>9</sup>[http://linuxwireless.org/en/developers/Documentation/mac80211/\\_\\_\\_v49.html](http://linuxwireless.org/en/developers/Documentation/mac80211/___v49.html)

<sup>10</sup><https://w1.fi/cgiit/hostap/plain/tests/hwsim/example-setup.txt>

<sup>11</sup>[http://www2.cs.siu.edu/~sharvey/code/cs441/cs441\\_lab.pdf](http://www2.cs.siu.edu/~sharvey/code/cs441/cs441_lab.pdf)

<sup>12</sup><http://upcommons.upc.edu/bitstream/handle/2099.1/19202/memoria.pdf?sequence=4>

<sup>13</sup><https://nims11.wordpress.com/2012/04/27/hostapd-the-linux-way-to-create-virtual-wifi-access-point/>

<sup>14</sup>[https://wiki.debian.org/WiFi/HowToUse#wpa\\_supPLICANT](https://wiki.debian.org/WiFi/HowToUse#wpa_supPLICANT)

<sup>15</sup>[https://www.kernel.org/doc/README/Documentation-networking-mac80211\\_hwsim-README](https://www.kernel.org/doc/README/Documentation-networking-mac80211_hwsim-README)

<sup>16</sup>[https://github.com/penberg/linux-kvm/tree/master/Documentation/networking/mac80211\\_hwsim](https://github.com/penberg/linux-kvm/tree/master/Documentation/networking/mac80211_hwsim)

<sup>17</sup><http://www.brianlinkletter.com/installing-debian-linux-in-a-virtualbox-virtual-machine/>

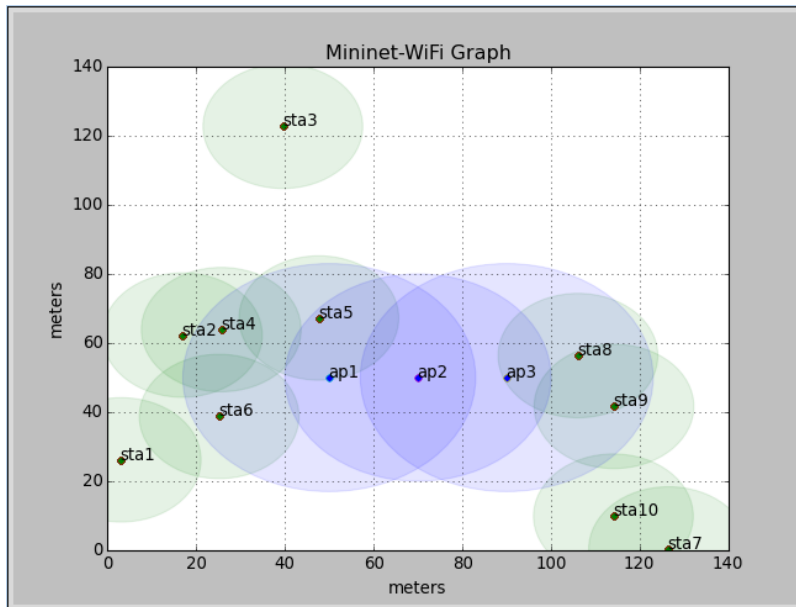


Figure 4.1: Mininet-WiFi Graph

#### 4.2.8 Set up the Mininet-WiFi VM

To ensure that the VM can display X applications such as Wireshark on your host computer's desktop, read through my post about [setting up the standard Mininet VM](#)<sup>18</sup> and set up the host-only network adapter, the X windows server, and your SSH software.

Now you can connect to the VM via SSH with X Forwarding enabled. In the example below, my host computer is t420 and the Mininet WiFi VM is named wifi.

```
t420:~$ ssh -X 192.168.52.101
wifi:~$
```

### 4.3 Mininet-WiFi Tutorial #1: One access point

The simplest network is the default topology, which consists of a wireless access point with two wireless stations. The access point is a switch connected to a controller. The stations are hosts.

This simple lab will allow us to demonstrate how to capture wireless control traffic and will demonstrate the way an OpenFlow-enabled access point handles WiFi traffic on the wlan interface.

#### 4.3.1 Capturing Wireless control traffic in Mininet-WiFi

To view wireless control traffic we must first start Wireshark:

```
wifi:~$ wireshark &
```

<sup>18</sup><http://www.brianlinkletter.com/set-up-mininet/>



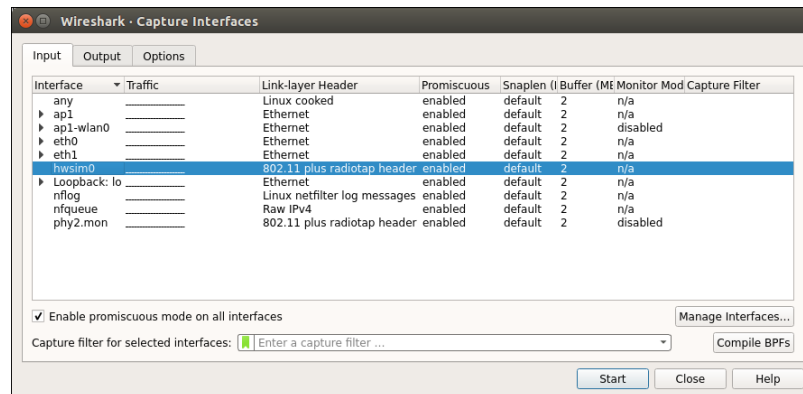


Figure 4.2: Start capture on hwsim0 interface

Then, start Mininet-WiFi with the default network scenario using the command below:

```
wifi:~$ sudo mn --wifi
```

Next, enable the hwsim0 interface. The hwsim0 interface is the software interface created by Mininet-WiFi that copies all wireless traffic to all the virtual wireless interfaces in the network scenario. It is the easiest way to monitor the wireless packets in Mininet-WiFi.

```
mininet-wifi> sh ifconfig hwsim0 up
```

Now, in Wireshark (figure 4.2, refresh the interfaces and then start capturing packets on the \*hwsim0\* interface. You should see wireless control traffic. Next, run a ping command:

```
mininet-wifi> sta1 ping sta2
```

In Wireshark (figure 4.3), see the wireless frames and the ICMP packets encapsulated in Wireless frames passing through the hwsim0 interface. Stop the ping command by pressing **Ctrl-C**. In this default setup, any flows created in the access point (that's if they're created – see below for more on this issue) will expire in 60 seconds.

#### 4.3.2 Wireless Access Points and OpenFlow

In this simple scenario, the access point has only one interface, `ap1-wlan0`. By default, stations associated with an access point connect in infrastructure mode so wireless traffic between stations must pass through the access point. If the access point works similarly to a switch in standard Mininet, we expect to see OpenFlow messages exchanged between the access point and the controller whenever the access point sees traffic for which it does not already have flows established.

To view OpenFlow packets, stop the Wireshark capture and switch to the loopback interface. Start capturing again on the `loopback` interface. Use the `OpenFlow_1.0` filter to view only OpenFlow messages.

Then, start some traffic running with the ping command and look at the OpenFlow messages captured in Wireshark.

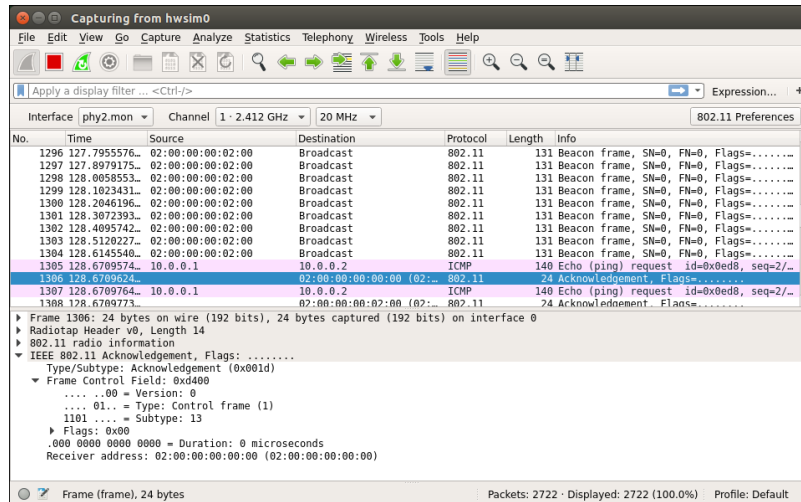


Figure 4.3: Wireshark capturing WiFi control traffic

```
mininet-wifi> sta1 ping sta2
```

I was expecting that the first ICMP packet generated by the ping command should be flooded to the controller, and the controller would set up a flows on the access point so the two stations could exchange packets. Instead, I found that the two stations were able to exchange packets immediately and the access point did not flood the ICMP packets to the controller. Only an ARP packet, which is in a broadcast frame, gets flooded to the controller and is ignored (figure 4.4).

Check to see if flows have been created in the access point:

```
mininet-wifi> dpctl dump-flows
*** ap1 -----
NXST_FLOW reply (xid=0x4):
```

We see that no flows have been created on the access point. How do the two access points communicate with each other?

I do not know the answer but I have an idea. My research indicates that OpenFlow-enabled switches (using OpenFlow 1.0 or 1.3) will reject "hairpin connections", which are flows that cause traffic to be sent out the same port in which it was received. A wireless access point, by design, receives and sends packets on the same wireless interface. Stations connected to the same wireless access point would require a "hairpin connection" on the access point to communicate with each other. I surmise that, to handle this issue, Linux treats the WLAN interface in each access point like the radio network `sta1-ap1-sta2` as if it is a "hub", where `ap1-wlan0` provides the "hub" functionality for data passing between `sta1` and `sta2`. `ap1-wlan0` switches packets in the wireless domain and will not bring a packet into the "Ethernet switch" part of access point `ap1` unless it must be switched to another interface on `ap1` other than back out `ap1-wlan0`.

### 4.3.3 Stop the tutorial

Stop the Mininet ping command by pressing Ctrl-C.

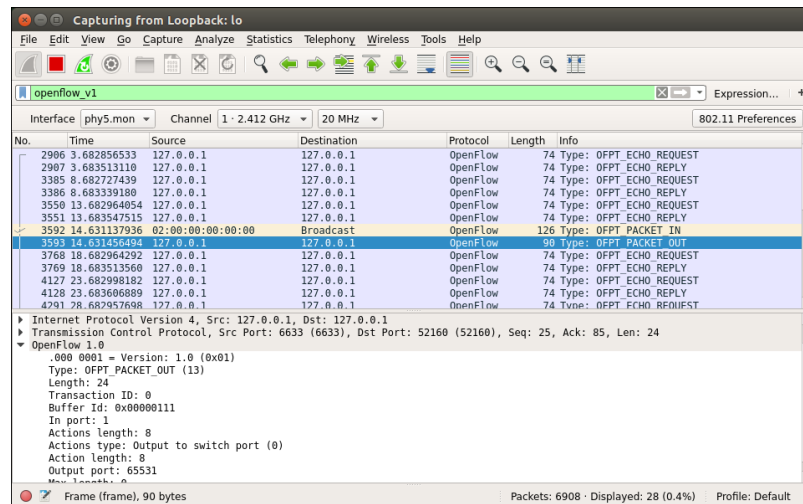


Figure 4.4: No OpenFlow messages passing to the controller

In the Wireshark window, stop capturing and quit Wireshark.

Stop Mininet-Wifi and clean up the system with the following commands:

```
mininet-wifi> exit
wifi:~$ sudo mn -c
```

## 4.4 Mininet-WiFi Tutorial #2: Multiple access points

When we create a network scenario with two or more wireless access points, we can show more of the functions available in Mininet-WiFi.

In this tutorial, we will create a linear topology with three access points, where one station is connected to each access point. Remember, you need to already know basic Mininet commands<sup>19</sup> to appreciate how we create topologies using the Mininet command line.

Run Mininet-Wifi and create a linear topology with three access points:

```
sudo mn --wifi --topo linear,3
```

From the output of the command, we can see how the network is set up and which stations are associated with which access points.

```
*** Creating network
*** Adding controller
*** Adding hosts and stations:
sta1 sta2 sta3
*** Adding switches and access point(s):
ap1 ap2 ap3
*** Adding links and associating station(s):
(ap2, ap1) (ap3, ap2) (sta1, ap1) (sta2, ap2) (sta3, ap3)
```

<sup>19</sup><http://mininet.org/walkthrough/>

```

*** Starting controller(s)
c0
*** Starting switches and access points
ap1 ap2 ap3 ...
*** Starting CLI:
mininet-wifi>

```

We can also verify the configuration using the Mininet CLI commands `net` and `dump`. For example, we can run the `net` command to see the connections between nodes:

```

mininet-wifi> net
sta1 sta1-wlan0:None
sta2 sta2-wlan0:None
sta3 sta3-wlan0:None
ap1 lo:  ap1-eth1:ap2-eth1
ap2 lo:  ap2-eth1:ap1-eth1 ap2-eth2:ap3-eth1
ap3 lo:  ap3-eth1:ap2-eth2
c0

```

From the `net` command above, we see that `ap1`, `ap2`, and `ap3` are connected together in a linear fashion by Ethernet links. But, we do not see any information about to which access point each station is connect. This is because they are connected over a "radio" interface so we need to run the `iw` command at each station to observe to which access point each is associated.

To check which access points are "visible" to each station, use the `iw scan` command:

```

mininet-wifi> sta1 iw dev sta1-wlan0 scan | grep ssid
SSID: ssid_ap1
SSID: ssid_ap2
SSID: ssid_ap3

```

Verify the access point to which each station is currently connected with the `iw link` command. For example, to see the access point to which station `sta1` is connected, use the following command:

```

mininet-wifi> sta1 iw dev sta1-wlan0 link
Connected to 02:00:00:00:03:00 (on sta1-wlan0)
    SSID: ssid_ap1
    freq: 2412
    RX: 1853238 bytes (33672 packets)
    TX: 7871 bytes (174 packets)
    signal: -30 dBm
    tx bitrate: 54.0 MBit/s

    bss flags:          short-slot-time
    dtim period:       2
    beacon int:        100
mininet-wifi>

```

#### 4.4.1 A simple mobility scenario

In this example, each station is connected to a different wireless access point. We can use the `iw` command to change which access point to which each station is connected.

Note: The `iw` commands may be used in static scenarios like this but should not be used when Mininet-WiFi automatically assigns associations in more realistic mobility scenarios. We'll discuss how Mininet-WiFi handles real mobility and how to use `iw` commands with Mininet-WiFi later in this post.

Let's decide we want `sta1`, which is currently associated with `ap1`, to change its association to `ap2`. Manually switch the `sta1` association from `ap1` (which is `ssid_ap1`) to `ap2` (which is `ssid_ap2`) using the following commands:

```
mininet-wifi> sta1 iw dev sta1-wlan0 disconnect
mininet-wifi> sta1 iw dev sta1-wlan0 connect ssid_ap2
```

Verify the change with the `iw link` command:

```
mininet-wifi> sta1 iw dev sta1-wlan0 link
    Connected to 02:00:00:00:04:00 (on sta1-wlan0)
        SSID: ssid_ap2
        freq: 2412
        RX: 112 bytes (4 packets)
        TX: 103 bytes (2 packets)
        signal: -30 dBm
        tx bitrate: 1.0 MBit/s

        bss flags:          short-slot-time
        dtim period:       2
        beacon int:        100
mininet-wifi>
```

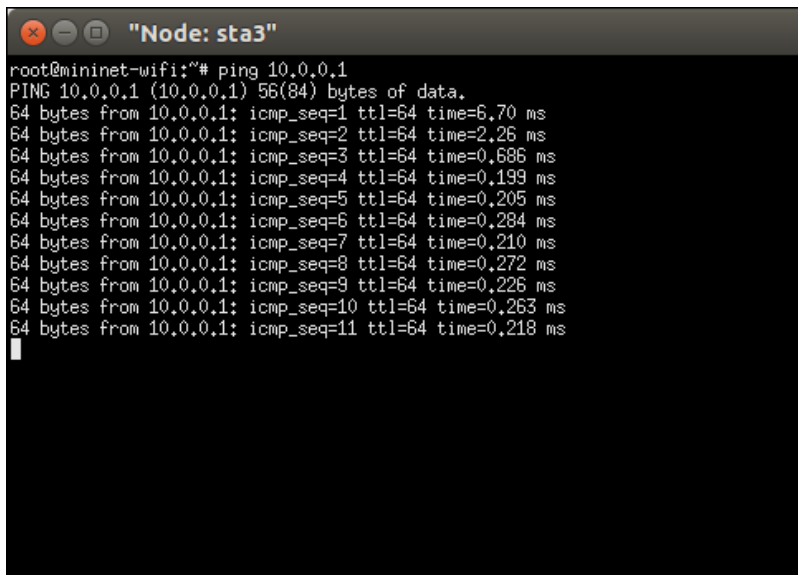
We see that `sta1` is now associated with `ap2`. So we've demonstrated a basic way to make stations mobile, where they switch their association from one access point to another.

#### 4.4.2 OpenFlow flows in a mobility scenario

Now let's see how the Mininet reference controller handles this simple mobility scenario. We need to get some traffic running from `sta1` to `sta3` in a way that allows us to access the Mininet-WiFi command line. We'll run the `ping` command in an `xterm` window on `sta3`.

First, check the IP addresses on `sta1` and `sta3` so we know which parameters to use in our test. The easiest way to see all IP addresses is to run the `dump` command:

```
mininet-wifi> dump
<Host sta1: sta1-wlan0:10.0.0.1 pid=7091>
<Host sta2: sta2-wlan0:10.0.0.2 pid=7094>
<Host sta3: sta3-wlan0:10.0.0.3 pid=7097>
<OVSSwitch ap1: lo:127.0.0.1,ap1-eth1:None pid=7106>
<OVSSwitch ap2: lo:127.0.0.1,ap2-eth1:None,ap2-eth2:None pid=7110>
```



```

root@mininet-wifi:~# ping 10.0.0.1
PING 10.0.0.1 (10.0.0.1) 56(84) bytes of data:
64 bytes from 10.0.0.1: icmp_seq=1 ttl=64 time=6.70 ms
64 bytes from 10.0.0.1: icmp_seq=2 ttl=64 time=2.26 ms
64 bytes from 10.0.0.1: icmp_seq=3 ttl=64 time=0.686 ms
64 bytes from 10.0.0.1: icmp_seq=4 ttl=64 time=0.199 ms
64 bytes from 10.0.0.1: icmp_seq=5 ttl=64 time=0.205 ms
64 bytes from 10.0.0.1: icmp_seq=6 ttl=64 time=0.284 ms
64 bytes from 10.0.0.1: icmp_seq=7 ttl=64 time=0.210 ms
64 bytes from 10.0.0.1: icmp_seq=8 ttl=64 time=0.272 ms
64 bytes from 10.0.0.1: icmp_seq=9 ttl=64 time=0.226 ms
64 bytes from 10.0.0.1: icmp_seq=10 ttl=64 time=0.263 ms
64 bytes from 10.0.0.1: icmp_seq=11 ttl=64 time=0.218 ms

```

Figure 4.5: xterm window on sta3

```

<OVSSwitch ap3: lo:127.0.0.1,ap3-eth1:None pid=7114>
<Controller c0: 127.0.0.1:6633 pid=7080>
mininet-wifi>

```

So we see that sta1 has IP address 10.0.0.1 and sta3 has IP address 10.0.0.3. Next, start an xterm window on sta3:

```
mininet-wifi> xterm sta3
```

This opens an xterm window (figure 4.5 from sta3).

In that window, run the following command to send ICMP messages from sta3 to sta1:

```
root@mininet-wifi:~# ping 10.0.0.1
```

Since these packets will be forwarded by the associated access points out a port other than the port on which the packets were received, the access points will operate like normal OpenFlow-enabled switches. Each access point will forward the first ping packet it receives in each direction to the Mininet reference controller. The controller will set up flows on the access points to establish a connection between the stations sta1 and sta3.

If we run Wireshark (figure 4.6) and enable packet capture on the Loopback interface, then filter using with of (for Ubuntu 14.04) or openflow\_v1 (for Ubuntu 15.10 and later), we will see OpenFlow messages passing to and from the controller. Now, in the Mininet CLI, check the flows on each switch with the `dpctl dump-flows` command.

```

mininet-wifi> dpctl dump-flows
*** ap1 -----
NXST_FLOW reply (xid=0x4):
*** ap2 -----

```

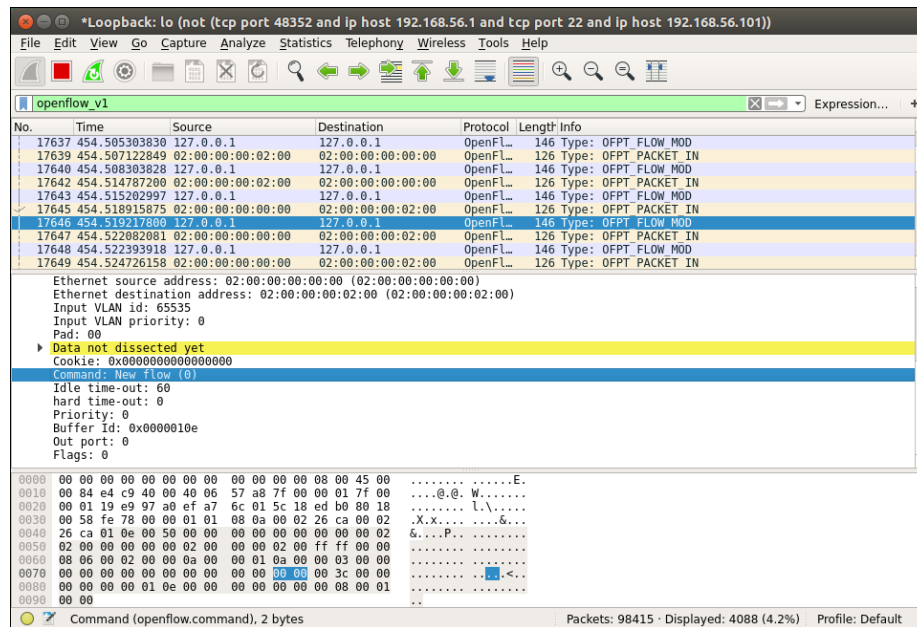


Figure 4.6: Wireshark capturing OpenFlow messages

```

NXST_FLOW reply (xid=0x4):
idle_timeout=60, idle_age=0,
→ priority=65535,arp,in_port=2,vlan_tci=0x0000,dl_src=02:00:00:00:02:00,
→ dl_dst=02:00:00:00:00:00,arp_spa=10.0.0.3,arp_tpa=10.0.0.1,arp_op=2
→ actions=output:3
cookie=0x0, duration=1068.17s, table=0, n_packets=35, n_bytes=1470,
→ idle_timeout=60, idle_age=0,
→ priority=65535,arp,in_port=3,vlan_tci=0x0000,dl_src=02:00:00:00:00:00,
→ dl_dst=02:00:00:00:02:00,arp_spa=10.0.0.1,arp_tpa=10.0.0.3,arp_op=1
→ actions=output:2
cookie=0x0, duration=1073.174s, table=0, n_packets=1073,
→ n_bytes=105154, idle_timeout=60, idle_age=0,
→ priority=65535,icmp,in_port=3,vlan_tci=0x0000,dl_src=02:00:00:00:00:00,
→ dl_dst=02:00:00:00:02:00,nw_src=10.0.0.1,nw_dst=10.0.0.3,nw_tos=0,
→ icmp_type=0,icmp_code=0 actions=output:2
cookie=0x0, duration=1073.175s, table=0, n_packets=1073,
→ n_bytes=105154, idle_timeout=60, idle_age=0,
→ priority=65535,icmp,in_port=2,vlan_tci=0x0000,dl_src=02:00:00:00:02:00,
→ dl_dst=02:00:00:00:00:00,nw_src=10.0.0.3,nw_dst=10.0.0.1,nw_tos=0,
→ icmp_type=8,icmp_code=0 actions=output:3
*** ap3 -----
NXST_FLOW reply (xid=0x4):

```

```

cookie=0x0, duration=1068.176s, table=0, n_packets=35, n_bytes=1470,
↳ idle_timeout=60, idle_age=0,
↳ priority=65535,arp,in_port=2,vlan_tci=0x0000,dl_src=02:00:00:00:02:00,
↳ dl_dst=02:00:00:00:00:00,arp_spa=10.0.0.3,arp_tpa=10.0.0.1,arp_op=2
↳ actions=output:1
idle_timeout=60, idle_age=0,
↳ priority=65535,arp,in_port=1,vlan_tci=0x0000,dl_src=02:00:00:00:00:00,
↳ dl_dst=02:00:00:00:02:00,arp_spa=10.0.0.1,arp_tpa=10.0.0.3,arp_op=1
↳ actions=output:2
cookie=0x0, duration=1073.182s, table=0, n_packets=1073,
↳ n_bytes=105154, idle_timeout=60, idle_age=0,
↳ priority=65535,icmp,in_port=1,vlan_tci=0x0000,dl_src=02:00:00:00:00:00,
↳ dl_dst=02:00:00:00:02:00,nw_src=10.0.0.1,nw_dst=10.0.0.3,nw_tos=0,
↳ icmp_type=0,icmp_code=0 actions=output:2
cookie=0x0, duration=1073.185s, table=0, n_packets=1073,
↳ n_bytes=105154, idle_timeout=60, idle_age=0,
↳ priority=65535,icmp,in_port=2,vlan_tci=0x0000,dl_src=02:00:00:00:02:00,
↳ dl_dst=02:00:00:00:00:00,nw_src=10.0.0.3,nw_dst=10.0.0.1,nw_tos=0,
↳ icmp_type=8,icmp_code=0 actions=output:1
mininet-wifi>

```

We see flows set up on ap2 and ap3, but not on ap1. This is because sta1 is connected to ap2 and sta3 is connected to ap3 so all traffic is passing through only ap2 and ap3. What will happen if sta1 moves back to ap1? Move sta1 back to access point ap1 with the following commands:

```

mininet-wifi> sta1 iw dev sta1-wlan0 disconnect
mininet-wifi> sta1 iw dev sta1-wlan0 connect ssid_ap1

```

The ping command running on sta3 stops working. We see no more pings completed.

In this case, access points ap2 and ap3 already have flows for ICMP messages coming from sta3 so they just keep sending packets towards the ap2-wlan0 interface to reach where they think sta1 is connected. Since ping messages never get to sta1 in its new location, the access point ap1 never sees any ICMP traffic so does not request any flow updates from the controller.

Check the flow tables in the access points again:

```

mininet-wifi> dpctl dump-flows
*** ap1 -----
NXST_FLOW reply (xid=0x4):
cookie=0x0, duration=40.959s, table=0, n_packets=1, n_bytes=42,
↳ idle_timeout=60, idle_age=40,
↳ priority=65535,arp,in_port=1,vlan_tci=0x0000,dl_src=02:00:00:00:02:00,
↳ dl_dst=02:00:00:00:00:00,arp_spa=10.0.0.3,arp_tpa=10.0.0.1,arp_op=1
↳ actions=output:2
cookie=0x0, duration=40.958s, table=0, n_packets=1, n_bytes=42,
↳ idle_timeout=60, idle_age=40,
↳ priority=65535,arp,in_port=2,vlan_tci=0x0000,dl_src=02:00:00:00:00:00,
↳ dl_dst=02:00:00:00:02:00,arp_spa=10.0.0.1,arp_tpa=10.0.0.3,arp_op=2
↳ actions=output:1

```



```

*** ap2 -----
NXST_FLOW reply (xid=0x4):
  cookie=0x0, duration=40.968s, table=0, n_packets=1, n_bytes=42,
    ↪ idle_timeout=60, idle_age=40,
    ↪ priority=65535,arp,in_port=2,vlan_tci=0x0000,dl_src=02:00:00:00:02:00,
    ↪ dl_dst=02:00:00:00:00:00,arp_spa=10.0.0.3,arp_tpa=10.0.0.1,arp_op=1
    ↪ actions=output:1
  cookie=0x0, duration=40.964s, table=0, n_packets=1, n_bytes=42,
    ↪ idle_timeout=60, idle_age=40,
    ↪ priority=65535,arp,in_port=1,vlan_tci=0x0000,dl_src=02:00:00:00:00:00,
    ↪ dl_dst=02:00:00:00:02:00,arp_spa=10.0.0.1,arp_tpa=10.0.0.3,arp_op=2
    ↪ actions=output:2
  cookie=0x0, duration=1214.279s, table=0, n_packets=1214,
    ↪ n_bytes=118972, idle_timeout=60, idle_age=0,
    ↪ priority=65535,icmp,in_port=2,vlan_tci=0x0000,dl_src=02:00:00:00:02:00,
    ↪ dl_dst=02:00:00:00:00:00,nw_src=10.0.0.3,nw_dst=10.0.0.1,nw_tos=0,
    ↪ icmp_type=8,icmp_code=0 actions=output:3
*** ap3 -----
NXST_FLOW reply (xid=0x4):
  cookie=0x0, duration=40.978s, table=0, n_packets=1, n_bytes=42,
    ↪ idle_timeout=60, idle_age=40,
    ↪ priority=65535,arp,in_port=2,vlan_tci=0x0000,dl_src=02:00:00:00:02:00,
    ↪ dl_dst=02:00:00:00:00:00,arp_spa=10.0.0.3,arp_tpa=10.0.0.1,arp_op=1
    ↪ actions=output:1
  cookie=0x0, duration=40.971s, table=0, n_packets=1, n_bytes=42,
    ↪ idle_timeout=60, idle_age=40,
    ↪ priority=65535,arp,in_port=1,vlan_tci=0x0000,dl_src=02:00:00:00:00:00,
    ↪ dl_dst=02:00:00:00:02:00,arp_spa=10.0.0.1,arp_tpa=10.0.0.3,arp_op=2
    ↪ actions=output:2
  cookie=0x0, duration=1214.288s, table=0, n_packets=1214,
    ↪ n_bytes=118972, idle_timeout=60, idle_age=0,
    ↪ priority=65535,icmp,in_port=2,vlan_tci=0x0000,dl_src=02:00:00:00:02:00,
    ↪ dl_dst=02:00:00:00:00:00,nw_src=10.0.0.3,nw_dst=10.0.0.1,nw_tos=0,
    ↪ icmp_type=8,icmp_code=0 actions=output:1
mininet-wifi>

```

The controller sees some LLC messages from sta1 but does not recognize that sta1 has moved to a new access point, so it does nothing. Since the controller does not modify any flows in the access points, none of the ICMP packets still being generated by sta3 will reach sta1 so it cannot reply. This situation will remain as long as the access points ap2 and ap3 continue to see ICMP packets from sta3, which keeps the old flow information alive in their flow tables.

One "brute force" way to resolve this situation is to delete the flows on the switches. In this simple example, it's easier to just delete all flows. Delete the flows in the access points using the command below:

```
mininet-wifi> dpctl del-flows
```

Now the ping command running in the xterm window on sta3 should show that pings are being completed again.

Once all flows were deleted, ICMP messages received by the access points do not match any existing flows so the access points communicate with the controller to set up new flows. If we dump the flows we see that the ICMP packets passing between sta3 and sta1 are now traversing across all three access points.

```
mininet-wifi> dpctl dump-flows
*** ap1 -----
NXST_FLOW reply (xid=0x4):
  cookie=0x0, duration=10.41s, table=0, n_packets=11, n_bytes=1078,
    ↳ idle_timeout=60, idle_age=0,
    ↳ priority=65535,icmp,in_port=2,vlan_tci=0x0000,dl_src=02:00:00:00:00:00,
    ↳ dl_dst=02:00:00:00:02:00,nw_src=10.0.0.1,nw_dst=10.0.0.3,nw_tos=0,
    ↳ icmp_type=0,icmp_code=0 actions=output:1
  cookie=0x0, duration=9.41s, table=0, n_packets=10, n_bytes=980,
    ↳ idle_timeout=60, idle_age=0,
    ↳ priority=65535,icmp,in_port=1,vlan_tci=0x0000,dl_src=02:00:00:00:02:00,
    ↳ dl_dst=02:00:00:00:00:00,nw_src=10.0.0.3,nw_dst=10.0.0.1,nw_tos=0,
    ↳ icmp_type=8,icmp_code=0 actions=output:2
*** ap2 -----
NXST_FLOW reply (xid=0x4):
  cookie=0x0, duration=10.414s, table=0, n_packets=11, n_bytes=1078,
    ↳ idle_timeout=60, idle_age=0,
    ↳ priority=65535,icmp,in_port=1,vlan_tci=0x0000,dl_src=02:00:00:00:00:00,
    ↳ dl_dst=02:00:00:00:02:00,nw_src=10.0.0.1,nw_dst=10.0.0.3,nw_tos=0,
    ↳ icmp_type=0,icmp_code=0 actions=output:2
  cookie=0x0, duration=9.417s, table=0, n_packets=10, n_bytes=980,
    ↳ idle_timeout=60, idle_age=0,
    ↳ priority=65535,icmp,in_port=2,vlan_tci=0x0000,dl_src=02:00:00:00:02:00,
    ↳ dl_dst=02:00:00:00:00:00,nw_src=10.0.0.3,nw_dst=10.0.0.1,nw_tos=0,
    ↳ icmp_type=8,icmp_code=0 actions=output:1
*** ap3 -----
NXST_FLOW reply (xid=0x4):
  cookie=0x0, duration=10.421s, table=0, n_packets=11, n_bytes=1078,
    ↳ idle_timeout=60, idle_age=0,
    ↳ priority=65535,icmp,in_port=1,vlan_tci=0x0000,dl_src=02:00:00:00:00:00,
    ↳ dl_dst=02:00:00:00:02:00,nw_src=10.0.0.1,nw_dst=10.0.0.3,nw_tos=0,
    ↳ icmp_type=0,icmp_code=0 actions=output:2
  cookie=0x0, duration=9.427s, table=0, n_packets=10, n_bytes=980,
    ↳ idle_timeout=60, idl_age=0,
    ↳ priority=65535,icmp,in_port=2,vlan_tci=0x0000,dl_src=02:00:00:00:02:00,
    ↳ dl_dst=02:00:00:00:00:00,nw_src=10.0.0.3,nw_dst=10.0.0.1,nw_tos=0,
    ↳ icmp_type=8,icmp_code=0 actions=output:1
mininet-wifi>
```

We have shown how the Mininet reference controller works in Mininet-WiFi. The Mininet

reference controller does not have the ability to detect when a station moves from one access point to another. When this happens, we must delete the existing flows so that new flows can be created. We will need to use a more advanced remote controller, such as OpenDaylight, to enable station mobility but that is a topic outside the scope of this post.

#### 4.4.3 Stop the tutorial

Stop the Mininet ping command by pressing Ctrl-C. In the Wireshark window, stop capturing and quit Wireshark. Stop Mininet-Wifi and clean up the system with the following commands:

```
mininet-wifi> exit
wifi:~$ sudo mn -c
```

### 4.5 Mininet-WiFi Tutorial #3: Python API and scripts

Mininet provides a Python API so users can create simple Python scripts that will set up custom topologies. Mininet-WiFi extends this API to support a wireless environment.

When you use the normal Mininet `mn` command with the `-wifi` option to create Mininet-WiFi topologies, you do not have access to most of the extended functionality provided in Mininet-WiFi. To access features that allow you to emulate the behavior of nodes in a wireless LAN, you need to use the Mininet-Wifi extensions to the Mininet Python API.

#### 4.5.1 The Mininet-WiFi Python API

The Mininet-WiFi developers added new classes to Mininet to support emulation of nodes in a wireless environment. Mininet-WiFi adds `addStation` and `addAccessPoint` methods, and a modified `addLink` method to define the wireless environment.

If you are just beginning to write scripts for Mininet-WiFi, you can use the example scripts as a starting point. The Mininet-WiFi developers created example scripts that show how to use most of the features in Mininet-WiFi. In all of the tutorials I show below, I started with an example script and modified it.

Mininet-Wifi example scripts are in the `/mininet-wifi/examples` directory.

#### 4.5.2 Basic station and access point methods

In a simple scenario, you may add a station and an access point with the following methods in a Mininet-WiFi Python script:

Add a new station named `sta1`, with all parameters set to default values:

```
net.addStation( 'sta1' )
```

Add a new access point named `ap1`, with SSID `ap1-ssid`, and all other parameters set to default values:

```
net.addAccessPoint( 'ap1', ssid='new_ssid' )
```

Add a wireless association between station and access point, with default values for link attributes:

```
net.addLink( ap1, sta1 )
```

For more complex scenarios, more parameters are available for each method. You may specify the MAC address, IP address, location in three dimensional space, radio range, and more. For example, the following code defines an access point and a station, and creates an association (a wireless connection) between the two nodes and applies some traffic control parameters to the connection to make it more like a realistic radio environment, adding badwidth restrictions, an error rate, and a propagation delay:

Add a station and specify the wireless encryption method, the station MAC address, IP address, and position in virtual space:

```
net.addStation( 'sta1', passwd='123456789a', encrypt='wpa2',
→ mac='00:00:00:00:00:02', ip='10.0.0.2/8', position='50,30,0' )
```

Add an access point and specify the wireless encryption method, SSID, wireless mode, channel, position, and radio range:

```
net.addAccessPoint( 'ap1', passwd='123456789a', encrypt='wpa2', ssid=
→ 'ap1-ssid', mode= 'g', channel= '1', position='30,30,0', range=30 )
```

To activate association control in a static network, you may use the `*setAssociationCtrl*` method, which makes Mininet-WiFi automatically choose which access point a station will connect to based on the range between stations and access points. For example, use the following method to use the `*strongest signal first*` when determining connections between station and access points:

```
net.setAssociationCtrl( 'ssf' )
```

### 4.5.3 Classic Mininet API

The Mininet WiFi Python API still supports the standard Mininet node types – switches, hosts, and controllers. For example:

Add a host. Note that the station discussed above is a type of host node with a wireless interface instead of an Ethernet interface.

```
net.addHost( 'h1' )
```

Add a switch. Note that the access point discussed above is a type of switch that has one wireless interface (`*wlan0*`) and any number of Ethernet interfaces (up to the maximum supported by your installed version of Open vSwitch).

```
net.addSwitch( 's1' )
```

Add an Ethernet link between two nodes. Note that if you use `*addLink*` to connect two access points together (and are using the default Infrastructure mode), Mininet-WiFi creates an Ethernet link between them.

```
net.addLink( s1, h1 )
```

Add a controller:

```
net.addController( 'c0' )
```

Using the Python API, you may build a topology that includes hosts, switches, stations, access points, and multiple controllers.

### 4.5.4 Example

In the example below, I created a Python program that will set up two stations connected to two access points, and set node positions and radio range so that we can see how these properties affect the emulated network.

```

1  #!/usr/bin/python
2
3  from mininet.node import Controller
4  from mininet.log import setLogLevel, info
5  from mn_wifi.net import Mininet_wifi
6  from mn_wifi.node import OVSKernelAP
7  from mn_wifi.cli import CLI_wifi
8
9
10 def topology():
11
12     net = Mininet_wifi( controller=Controller, accessPoint=OVSKernelAP )
13
14     info( "*** Creating nodes\n" )
15     ap1 = net.addAccessPoint( 'ap1', ssid= 'ssid-ap1', mode= 'g', channel= '1',
16                               position='10,30,0', range='20' )
17     ap2 = net.addAccessPoint( 'ap2', ssid= 'ssid-ap2', mode= 'g', channel= '6',
18                               position='50,30,0', range='20' )
19     sta1 = net.addStation( 'sta1', mac='00:00:00:00:00:01', ip='10.0.0.1/8',
20                             position='10,20,0' )
21     sta2 = net.addStation( 'sta2', mac='00:00:00:00:00:02', ip='10.0.0.2/8',
22                             position='50,20,0' )
23     c1 = net.addController( 'c1', controller=Controller )
24
25     info( "*** Configuring wifi nodes" )
26     net.configureWifiNodes()
27
28     net.plotGraph( max_x=60, max_y=60 )
29
30     info( "*** Enabling association control (AP)\n" )
31     net.setAssociationCtrl( 'ssf' )
32
33     info( "*** Creating links and associations\n" )
34     net.addLink( ap1, ap2 )
35     net.addLink( ap1, sta1 )
36     net.addLink( ap2, sta2 )
37
38     info( "*** Starting network\n" )
39     net.build()
40     c1.start()
41     ap1.start( [c1] )
42     ap2.start( [c1] )
43
44     info( "*** Running CLI\n" )
45     CLI_wifi( net )
46
47     info( "*** Stopping network\n" )
48     net.stop()
49
50 if __name__ == '__main__':
51     setLogLevel( 'info' )

```

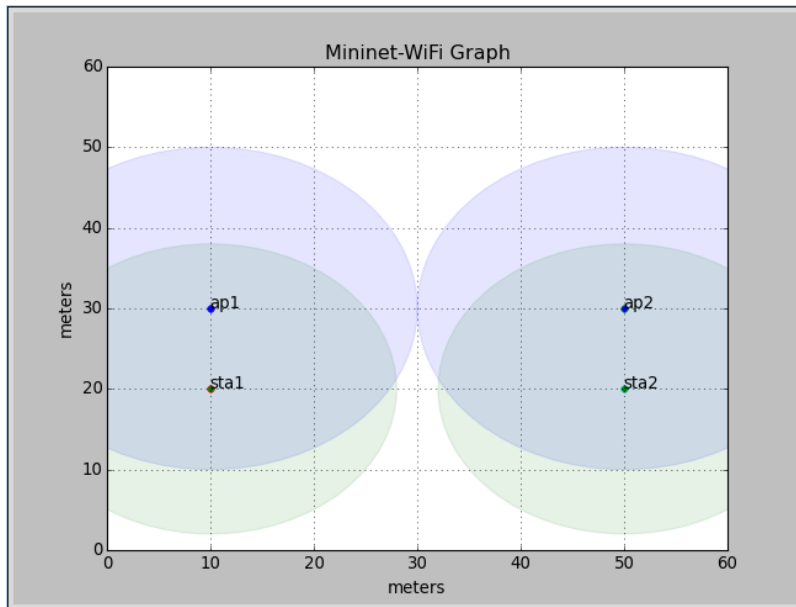


Figure 4.7: The position-test.py script running

48

```
topology ()
```

```
codes/position-test.py
```

I saved the file with the name `position-test.py` and made it executable.

#### 4.5.5 Working at runtime

Mininet-WiFi python scripts may be run from the command line by running the script directly, or by calling it as part of a Python command. The only difference is how the path is stated. For example:

```
wifi:~/scripts $ sudo ./position-test.py
```

or,

```
wifi:~$ sudo python position-test.py
```

The `position-test.py` script will set open the Mininet-WiFi graph window and show the locations of each wireless node in space (figure 4.7, and the range attribute of each node.

While the scenario is running, we can query information about the network from either the Mininet-WiFi command line or from the Python interpreter and we can log into running nodes to gather information or make configuration changes.

#### 4.5.6 Mininet-WiFi CLI

The Python script `position-test.py` places nodes in specific positions. When the scenario is running, we can use the Mininet-WiFi command line interface (CLI) commands to can check the geometric relationship between nodes in space, and information about each node.

### 4.5.7 Position

The `position` CLI command outputs the location of a node in virtual space as measured by three values, one for each of the vertices X, Y, and Z.

Suppose we want to know the position of the access point `ap1` in the network scenario's virtual space. We may use the `position` CLI command to view a node's position:

```
mininet-wifi> py ap1.params['position']
```

We may also check the position of the station `sta2`:

```
mininet-wifi> py sta.params['position']
```

### 4.5.8 Distance

The `distance` CLI command tells us the distance between two nodes. For example, we may check how far apart access point `ap1` and station `sta2` are from each other using the `distance` CLI command:

```
mininet-wifi> distance ap1 sta2
The distance between ap1 and sta2 is 41.23 meters
```

### 4.5.9 Mininet-WiFi Python runtime interpreter

In addition to the CLI, Mininet-WiFi supports running Python code directly at the command line using the `py` command. Simple, Python functions may be called to get additional information about the network, or to make simple changes while the scenario is running.

### 4.5.10 Getting network information

The examples I show below are useful for gathering information about stations and access points.

To see the range of an access point or station, call the `range` function. Call it using the name of the node followed by the function as shown below for access point `ap1`:

```
mininet-wifi> py ap1.params['range']
20
```

To see which station is associated with an access point (in this example `ap1`) call the `associatedStations` function:

```
mininet-wifi> py ap1.params['associatedStations']
[<Host sta1: sta1-wlan0:10.0.0.1 pid=3845> ]
```

To see which access point is associated with a station (in this example `sta1`) call the `associatedTo` key:

```
mininet-wifi>py sta1.params['associatedTo']
[<OVSSwitch ap1: lo:127.0.0.1,ap1-eth1:None pid=3862>]
```

You may also query the received signal strength indicator (`rss`), transmitted power (`txpower`), service set indicator (`ssid`), channel, and frequency of each wireless node using the Python interpreter.

As we can see, the output of Python functions is formatted as strings and numbers that may sometimes be hard to read. This is because these functions are built to support the program, not to be read by humans. However, if you which functions are available to be called at the Mininet-WiFi command line you will be able to get information you cannot get through the standard Mininet-WiFi CLI.

#### 4.5.11 Changing the network during runtime

Mininet-WiFi provides Python functions that can be used during runtime to make changes to node positions and associations. These functions are useful when we have a static setup and want to make arbitrary changes on demand. This makes it possible to do testing or demonstrations with carefully controlled scenarios.

To change the access point to which a station is associated (provided the access point is within range):

```
mininet-wifi> py sta1.moveAssociationTo('sta1-wlan0', ap1)
```

To move a station or access point in space to another coordinate position:

```
mininet-wifi> py sta1.setPosition('40,20,40')
```

To change the range of a station or access point:

```
mininet-wifi> py sta1.setRange(100)
```

The commands above will all impact which access points and which stations associate with each other. The behavior of the network will be different depending on whether association control is enabled or disabled in the `position-test.py` script.

#### 4.5.12 Running commands in nodes

When running a scenario, users may make configuration changes on nodes to implement some additional functionality. This can be done from the Mininet-WiFi command line by sending commands to the node's command shell. Start the command with the name of the node followed by a space, then enter the command to run on that node.

For example, to see information about the WLAN interface on a station named `sta1`, run the command:

```
mininet-wifi> sta1 iw dev sta1-wlan0 link
```

Another way to run commands on nodes is to open an `xterm` window on that node and enter commands in the `xterm` window. For example, to open an `xterm` window on station `*sta1*`, run the command:

```
mininet-wifi> xterm sta1
```



Running commands on nodes is standard Mininet feature but it is also an advanced topic. See the Mininet documentation<sup>20</sup> for more details. You can run simple commands such as `ping` or `iwconfig` but more advance commands may require you to mount private directories<sup>21</sup> for configuration or log files.

#### 4.5.13 Mininet-WiFi and shell commands

Mininet-WiFi manages the affect of range using code that calculates the ability of each node to connect with other nodes. However, Mininet-WiFi does not change the way networking works at the operating system level. So `iw` commands executed on nodes will override Mininet-WiFi and do not gather information generated by Mininet-WiFi about the network.

I suggest you do not rely on `iw` commands. For example, the `iw scan` command will still show that `sta1` can detect the SSIDs of all access points, even the access point `ap2` which should be out of range. The `iw link` command will show the same signal strength regardless of how far the station is from the access point, while the Mininet-WiFi `*info*` command will show the calculated signal strength based on the propagation model and distance between nodes.

For example, the `iw` command run on `sta1` shows received signal strength is -30 dBm. This never changes no matter how far the station is from the access point.

```
mininet-wifi> sta1 iw dev sta1-wlan0 link
Connected to 02:00:00:00:00:00 (on sta1-wlan0)
    SSID: ssid-ap1
    freq: 2412
    RX: 164628 bytes (2993 packets)
    TX: 775 bytes (10 packets)
    signal: -30 dBm
    tx bitrate: 6.0 MBit/s

    bss flags:      short-slot-time
    dtim period:    2
    beacon int:     100
```

The `info` command shows Mininet-WiFi's calculated signal strength received by the station is -43.11 dBm. This value will change if you reposition the station.

#### 4.5.14 Stop the tutorial

Stop Mininet-Wifi and clean up the system with the following commands:

```
mininet-wifi> exit
wifi:~$ sudo mn -c
```

## 4.6 Mininet-WiFi Tutorial #4: Mobility

The more interesting features provided by Mininet-WiFi support mobile stations moving around in virtual space. Mininet-Wifi provides new methods in its Python API, such as `startMobility`

<sup>20</sup><https://github.com/mininet/mininet/wiki/Documentation>

<sup>21</sup><https://github.com/mininet/mininet/wiki/Introduction-to-Mininet#important-shared-filesystem>

and Mobility, with which we may specify a wide variety of wireless LAN scenarios by controlling station movement, access point range, radio propagation models, and more.

In this tutorial, we will create a scenario where one station moves about in space, and where it changes which access point it connects to, based on which access point is the closest.

#### 4.6.1 Python API and mobility

The Mininet-WiFi Python API adds new methods that allow the user to create stations that move around in virtual space when an emulation scenario is running.

To move a station in a straight line, use the `net.startMobility` and `net.mobility` methods. See the example script `wifiMobility.py`. For example, to move a station from one position to another over a period of 60 seconds, add the following lines to your script:

```
net.startMobility( time=0 )
net.mobility( sta1, 'start', time=1, position='10,20,0' )
net.mobility( sta1, 'stop', time=59, position='30,50,0' )
net.stopMobility( time=60 )
```

Mininet-WiFi can also automatically move stations around based on predefined mobility models. See the example script `mobilityModel.py`. Available mobility models are: `RandomWalk`, `TruncatedLevyWalk`, `RandomDirection`, `RandomWayPoint`, `GaussMarkov`, `ReferencePoint`, and `TimeVariantCommunity`. For example, to move a station around in an area 60 meters by 60 meters with a minimum velocity of 0.1 meters per second and a maximum velocity of 0.2 meters per second, add the following line to your script:

```
net.setMobilityModel(time=0, model='RandomDirection', max_x=60,
    ↪ max_y=60, min_v=0.1, max_v=0.2)
```

Mininet-WiFi will automatically connect and disconnect stations to and from access points based on either calculated signal strength or load level. To use an association control method, you need to add the `AC` parameter in `net.setMobilityModel()`. For example, to switch access points based on the “least loaded first” criteria, add the following line in your script:

```
net.setMobilityModel(time=0, model='RandomWayPoint', max_x=140,
    ↪ max_y=140, min_v=0.7, max_v=0.9, AC='llf')
```

The valid values for the `AC` parameter are:

- llf (Least-Loaded-First)
- ssf (Strongest-Signal-First)

#### 4.6.2 Moving a station in virtual space

A simple way to demonstrate how Mininet-WiFi implements scenarios with mobile stations that hand off between access points is to create a script that moves one station across a path that passes by three access points.

The example below will create three access points – `ap1`, `ap2`, and `ap3` – arranged in a line at differing distances from each other. It also creates a host `h1` to serve as a test server and a mobile station `sta1` and moves `sta1` across space past all three access points.

```

1  #!/usr/bin/python
2
3  from mininet.node import Controller
4  from mininet.log import setLogLevel, info
5  from mn_wifi.net import Mininet_wifi
6  from mn_wifi.node import OVSKernelAP
7  from mn_wifi.cli import CLI_wifi
8
9
10 def topology():
11
12     net = Mininet_wifi( controller=Controller, accessPoint=OVSKernelAP )
13
14     info( "*** Creating nodes\n"
15     h1 = net.addHost( 'h1', mac='00:00:00:00:00:01', ip='10.0.0.1/8' )
16     sta1 = net.addStation( 'sta1', mac='00:00:00:00:00:02', ip='10.0.0.2/8',
17     range='20' )
18     ap1 = net.addAccessPoint( 'ap1', ssid='ap1-ssid', mode='g', channel='1',
19     position='30,50,0', range='30' )
20     ap2 = net.addAccessPoint( 'ap2', ssid='ap2-ssid', mode='g', channel='1',
21     position='90,50,0', range='30' )
22     ap3 = net.addAccessPoint( 'ap3', ssid='ap3-ssid', mode='g', channel='1',
23     position='130,50,0', range='30' )
24     cl = net.addController( 'c1', controller=Controller )
25
26     info( "*** Configuring wifi nodes\n"
27     net.configureWifiNodes()
28
29     info( "*** Associating and Creating links\n"
30     net.addLink( ap1, h1 )
31     net.addLink( ap1, ap2 )
32     net.addLink( ap2, ap3 )
33
34     net.plotGraph( max_x=160, max_y=160 )
35
36     net.startMobility( time=0, AC='ssf' )
37     net.mobility( sta1, 'start', time=20, position='1,50,0' )
38     net.mobility( sta1, 'stop', time=79, position='159,50,0' )
39     net.stopMobility( time=80 )
40
41     info( "*** Starting network\n"
42     net.build()
43     cl.start()
44     ap1.start( [cl] )
45     ap2.start( [cl] )
46     ap3.start( [cl] )
47
48     info( "*** Running CLI\n"
49     CLI_wifi( net )
50
51     info( "*** Stopping network\n"
52     net.stop()
53
54 if __name__ == '__main__':
55     setLogLevel( 'info' )
56     topology()

```

---

codes/line.py

Save the script and call in `line.py`. Make it executable, then run the command:

```
wifi:~$ sudo ./line.py
```

The Mininet-Wifi graph will appear (figure 4.8), showing the station and the access points.

The station `sta1` will sit still for 20 seconds, and then start to move across the graph from left to right for 60 seconds until it gets to the far side of the graph. The host `h1` and the virtual Ethernet connections between `h1`, `ap1` and between the three access points are not visible.

#### 4.6.3 Re-starting the scenario

This simple scenario has a discreet start and stop time so, if you wish to run it again, you need to quit Mininet-WiFi, and start the script again.

For example, suppose the scenario is at its end, where the station is now at the far right of the graph window. To stop and start it again, enter the following commands:

```
mininet-wifi> exit
wifi:~$ sudo mn -c
wifi:~$ sudo ./line.py
```

#### 4.6.4 More Python functions

When running a scenario with the mobility methods in the Python API, we have access to more information from Mininet-WiFi's Python functions. To see all access points that are within range of a station such as `sta1` at any time while the scenario is running, call the `apsInRange` function:

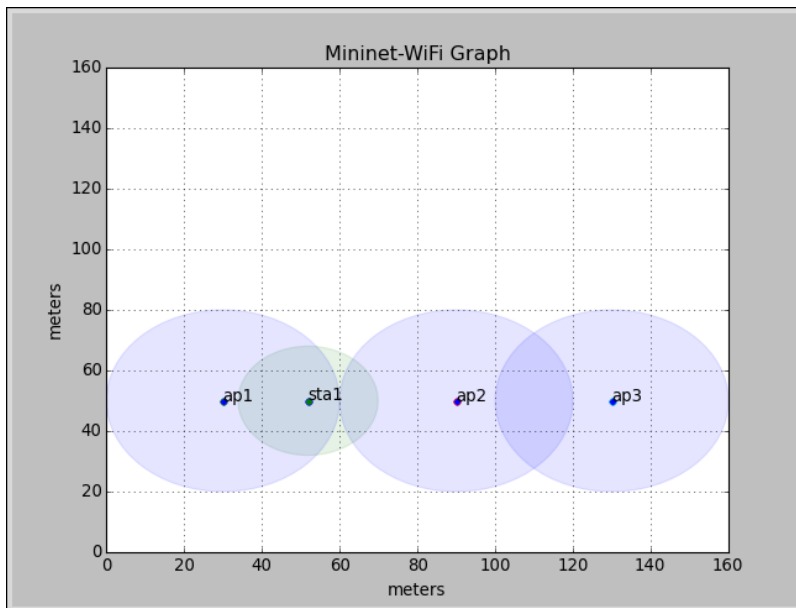


Figure 4.8: The `line.py` script running

```
mininet-wifi> py sta1.params['apsInRange']  
[<OVSSwitch ap1: 10:127.0.0.1,ap1-eth1:None pid=3862>]
```

#### 4.6.5 Test with iperf

To see how the system responds to traffic, run some data between host `h1` and station `sta1` when the scenario is started.

We have seen in previous examples how to use the `ping` program to create traffic. In this example, we will use the `iperf` program. First, start the `line.py` script again. Then start an `iperf` server on the station

```
mininet-wifi> sta1 iperf --server
```

Then open an `xterm` window on the host `h1`.

```
mininet-wifi> xterm h1
```

From the `xterm` window, we will start the `iperf` client command and create a stream of data between `h1` and `sta1`. On the `h1` `xterm`, run the command:

```
iperf --client 10.0.0.2 --time 60 --interval 2
```

Watch the `iperf` output as the station moves through the graph. When it passes from one access point to the next, the traffic will stop. To get the traffic running again, clear the flow tables in the access points. In the Mininet-WiFi CLI, run the command shown below:

```
mininet-wifi> dpctl del-flows
```

Traffic should start running again. As stated in Tutorial #2 above, we must clear flows after a hand off because the Mininet reference controller cannot respond correctly in a mobility scenario. The topic of configuring a remote controller to support a mobility scenario is outside the scope of this post. Clear the flows every time the station switches to the next access point.

#### 4.6.6 Stop the tutorial

Stop Mininet-Wifi and clean up the system with the following commands:

```
mininet-wifi> exit  
wifi:~$ sudo mn -c
```

## 4.7 Mininet-WiFi Tutorial #5: VANETs (Veicular Ad Hoc Networks)

### 4.7.1 Python API

The Mininet-WiFi python API add a new node that allow the user to create cars that move around in virtual space when an emulation scenario is running. The function `addCar()` defines this new node and you can see an example in the file `vanet.py` into `/examples`.

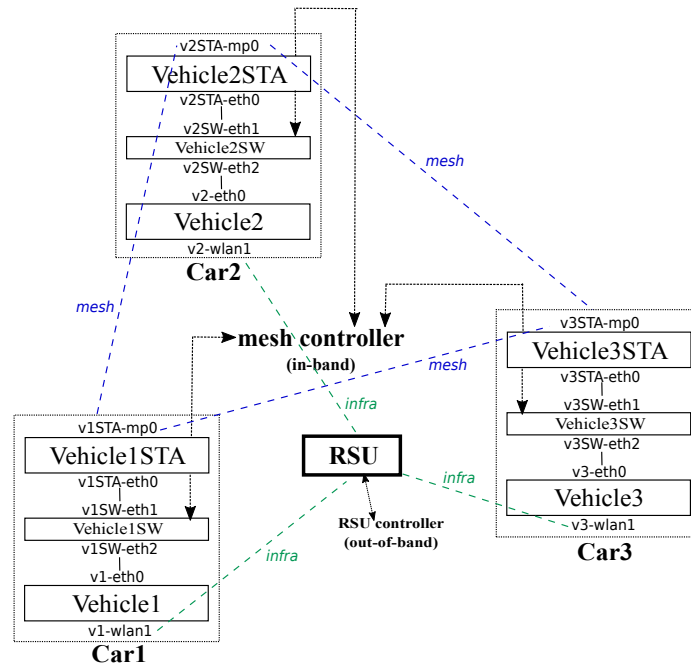


Figure 4.9: Node Car Architecture

#### 4.7.2 Node Car Architecture

The architecture of the node car is depicted in Figure 4.9. In order to allow OpenFlow controllers to control simple nodes, like a car, it was necessary to create an architecture with a switch. Thus, every packet that comes from VehicleX needs to pass-through VehicleXSW if the destination is a node in the wireless mesh network (like a Vehicle-to-Vehicle communication). The communication between cars and RSUs (Vehicle-to-Infrastructure), in turn, is done directly between VehicleX and the RSU.

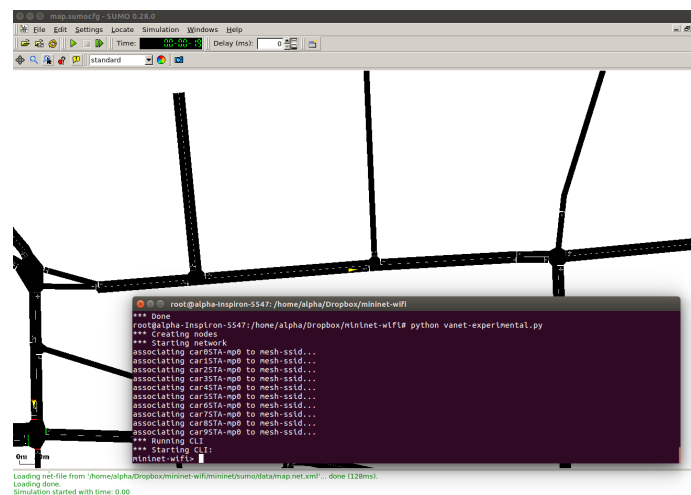


Figure 4.10: Integration with SUMO

### 4.7.3 Integration with SUMO (Simulation of Urban Mobility)

Mininet-WiFi already has some integration with SUMO. An example can be found in `/examples/sumo-experimental.py`.

## 4.8 Mininet-WiFi example scripts

The Mininet-WiFi developers created many example scripts that show examples of most of the API extensions they added to Mininet. They placed these example scripts in the folder `mininet-wifi/examples/`. Try running these scripts to see what they do and look at the code to understand how each feature is implemented using the Python API. Some interesting Mininet-WiFi example scripts are:

`adhoc` shows how to set up experiments with adhoc mode, where stations connect to each other without passing through an access point. `simplewifitopology` show the Python code that create the same topology as the default topology created by the `mn -wifi` command (two stations and one access point). `position.py` shows how to create a network where stations and access points are places in specific locations in virtual space. `mobility` and `mobilityModel` show how to move stations and how mobility models can be incorporated into scripts. `associationControl` shows how the different values of the AC parameter affect station handoffs to access points. `mesh.py` shows how to set up a mesh network of stations. `handover.py` shows how to create a simple mobility scenario where a station moves past two access points, causing the station to hand off from one to the other. `multipleWlan.py` shows how to create a station with more than one wireless LAN interface. `propagationModel.py` shows how to use propagation models that impact how stations and access points can communicate with each other over distance. `authentication.py` shows how to set up WiFi encryption and passwords on access points and stations.

## 4.9 Conclusion

The tutorials presented above work demonstrate many of the unique functions offered by Mininet-WiFi. Each tutorial revealed more functionality and we stopped at the point where we were able to emulate mobility scenario featuring a WiFi station moving in a straight line past several wireless access points.



Thank you Brian Linkletter for this helpful tutorial.







# Mininet-WiFi

Emulator for Software-Defined Wireless Networks

## 5. Guided exercises/demo

### 5.1 Guided exercises/demo

In general, you will learn in this chapter how the Mininet-WiFi wireless network emulator works. Guided exercises will be explored and pointers to source code for those interested in delving deeper into the system architecture will be also provided. The pointers include stretches of the code where the link (including the latency) can be customized.

#### 5.1.1 Activity 1: Warming up

First of all, you have to stop the network-manager:

```
sudo service network-manager stop
```

Then, create a simple topology with the command below:

```
sudo mn --wifi
```

The command above will start Mininet-WiFi and configure a small network with two stations, and one access point. Use the option `--topo` of `mn` command, and discover further the topology options. In Mininet-WiFi terminal (*mininet-wifi>*), execute the command nodes to observe the created network. Then, execute `iwconfig` to verify the association between the stations and `ap1`.

```
mininet-wifi>sta1 iwconfig
mininet-wifi>sta2 iwconfig
mininet-wifi>sta1 ping sta2
```

Then, disconnect `sta1` and confirm the disconnection:

```
mininet-wifi>sta1 iw dev sta1-wlan0 disconnect
mininet-wifi>sta1 iwconfig
mininet-wifi>sta1 ping sta2
```

Now, connect sta1 again:

```
mininet-wifi>sta1 iw dev sta1-wlan0 connect my-ssid
mininet-wifi>sta1 iwconfig
mininet-wifi>sta1 ping sta2
```

### 5.1.2 Activity 2: Loading Network Topologies

Running an example with the command below:

```
sudo python examples/position.py
```

Now, observe the position of sta1, sta2 and ap1:

```
mininet-wifi>py sta1.params['position']
mininet-wifi>py sta2.params['position']
mininet-wifi>py ap1.params['position']
```

Observe the signal power as well:

```
mininet-wifi>py sta1.params['rssi']
mininet-wifi>py sta2.params['rssi']
```

If you prefer, you can see the distance between two nodes with the following command:

```
mininet-wifi>distance sta1 ap1
mininet-wifi>distance sta1 sta2
```

**\* Question 2.1: What is the observed bandwidth between sta1 and sta2?**

*Tip: try iperf sta1 sta2*

Now, move sta1 to another position:

```
mininet-wifi>py sta1.setPosition('70,40,0')
```

**\* Question 2.2: What happened with the association between sta1 and ap1?**

*Tip: try sta1 iwconfig*

Finally, increase the signal range of ap1:

```
mininet-wifi>py ap1.setRange(60)
```

**\* Question 2.3: What happened with the association between sta1 and ap1 now?**

**\* Question 2.4: Now, observe the bandwidth between sta1 and sta2 again. What can we conclude?**

### 5.1.3 Activity 3: Customizing the Wireless Channel

Mininet-WiFi relies on the configuration of Linux TC (by default) to control the wireless channel properties, such as *bandwidth*, *packet loss*, *delay* and *latency*. Please see the equations in *mininet/wifiLink.py* (refer to *equationBW*, *equationLoss*, *equationDelay* and *equationLatency*. Those equations can be customized by calling *setChannelEquation()*, for example:

```
net.setChannelEquation(bw='value.rate * (1.1 ** -dist)', loss='(dist *
→ 2)/100', delay='(dist / 10) + 1', latency='2 + dist')
```

Alternatively, a new approach called *wmediumd* has been implemented, a wireless medium simulation tool for Linux, based on the *netlink* API implemented in the *mac80211\_hwsim* kernel driver. A couple of sample files that use *wmediumd* are available at */examples*.

**\* Question 3.1: Run *position.py* again and try *sta1 tc qdisc* before and after moving *sta1* to the new position. What do you can conclude about the configuration applied by *tc*?**

## 5.2 Further hands-on

(Further hands-on exercises proposed to be carried by the attendees on their own and at their pace. A number practical exercises will be proposed where mobility and/or distance (or poor signal) among mobile nodes may impact latency and bandwidth, consequently the communication between two nodes.)

### 5.2.1 Activity 4: Mobility

Open *examples/mobilityModel.py* with any text editor and change the velocity of stations as below:

```
from: net.startMobility(time=0, model='RandomDirection', max_x=100,
→ max_y=100, min_v=0.5, max_v=0.8)
to: net.startMobility(time=0, model='RandomDirection', max_x=100,
→ max_y=100, min_v=0.1, max_v=0.1)
```

Run *examples/mobilityModel.py* and change the signal range of *ap1*:

```
python examples/mobilityModel.py
mininet-wifi>py ap1.setRange(60)
```

Then, ping *sta1* and *sta2*:

```
mininet-wifi>sta1 ping sta2
```

**\* Question 4.1: What can you conclude about the latency?**

*Tip: you can issue *sta1 tc qdisc*, repeatedly, to see the values applied by *tc*.*

### 5.2.2 Activity 5: Received Signal Strength

Open *examples/position.py* with any text editor and add *sta3* at *position='10,10,10'* and set *max\_z=100* in order to plot a 3d graph. Then, run *examples/position.py*.

- \* **Question 5.1:** What is the received signal strength indicator (RSSI) observed from sta3?
- \* **Question 5.2:** What is the average ping response time between sta2 and sta1? And sta3 and sta1? *Note: define the number of packets to 10 (ping -c10).*

### 5.3 OpenFlow

You will learn with this activity some basic concepts of the OpenFlow protocol, such as idle/hard timeout and identify the impact of the mobility in the communication.

#### 5.3.1 Activity 6: Mobility and OpenFlow

First of all you have to get the code from <https://github.com/ramonfontes/reproducible-research/blob/master/mininet-wifi/ACROSS-Sweden-2017/handover.py> and then run it with the command below (the content of handover.py is also available in Code 9.1):

```
sudo python handover.py
```

Now, keep h1 pinging to sta1

```
mininet-wifi>h1 ping sta1
```

- \* **Question 6.1:** As you can see, h1 cannot reach sta1 when sta1 goes to ap2. Why? Two important commands should help you to answer this question:

```
mininet-wifi>links
mininet-wifi>sh ovs-ofctl dump-flows s3
```

*Tip: Observe both idle\_timeout and idle\_age.*

- \* **Question 6.2:** Now you know the answer to Question 6.1, how could sta1 be reached by h1?



# Mininet-WiFi

Emulator for Software-Defined Wireless Networks

## 6. Reproducible Research

The code and instructions used in this section can be found at <https://github.com/ramonfontes/reproducible-research>

### 6.1 SwitchOn 2015

*Extended abstract - Towards an Emulator for Software Defined Wireless Networks*

Firstly, you have to get the code at <https://github.com/ramonfontes/reproducible-research/blob/master/mininet-wifi/SWITCHON-2015>, then execute it:

```
sudo python allWirelessNetworksAroundUs.py
mininet-wifi> xterm st1 h1
```

On st1:

```
cvlc -vvv v4l2:///dev/video0 --input-slave=alsa://hw:1,0 --mtu 1000
→ --sout '#transcode{vcodec=mp4v,vb=800,scale=1,acodec=mpga,ab=128,channels=1}:
→ duplicate{dst=display,dst=rtp{sdp=rtsp://10.0.0.10:8080/helmet.sdp}}'
```

On h1:

```
cvlc rtsp://10.0.0.10:8080/helmet.sdp
```

### 6.2 SBRC 2016

*DEMO - Mininet-WiFi: Emulação de Redes Sem Fio Definidas por Software com suporte a Mobilidade*

Codes necessary to reproduce this paper are available at <https://github.com/ramonfontes/reproducible-research/tree/master/mininet-wifi/SBRC-2016> and <https://github.com/ramonfontes/reproducible-research/blob/master/mininet-wifi/SBRC-2016/wifiStationsAndHosts.py>.

### 6.2.1 Case 1 - Simple test

```
sudo mn --wifi
mininet-wifi>sta1 ping sta2
mininet-wifi>sta1 iwconfig
mininet-wifi>sta2 iwconfig
```

### 6.2.2 Case 2 (1/2) - Communication among stations and hosts/Verifying flow table

```
sudo python examples/wifiStationsAndHosts.py
mininet-wifi>nodes
mininet-wifi>sh ovs-ofctl dump-flows ap1
mininet-wifi>sta1 ping h3
mininet-wifi>sh ovs-ofctl dump-flows ap1
```

### 6.2.3 Case 2 (2/2) - Changing the controller (from reference to external controller)

Open the code examples/wifiStationsAndHosts.py and make the following changes:

```
from: net = Mininet( controller=Controller, link=TCLink,
    ↪ switch=OVSKernelSwitch )
to: net = Mininet( controller=RemoteController, link=TCLink,
    ↪ switch=OVSKernelSwitch )
from: c0 = net.addController('c0', controller=Controller, ip='127.0.0.1' )
to: c0 = net.addController('c0', controller=RemoteController, ip='127.0.0.1'
    ↪ )
sudo python examples/wifiStationsAndHosts.py
mininet-wifi>sta1 ping h3 #Why there is no communication?
```

### 6.2.4 Case 3 - Handover

```
sudo python examples/handover.py
mininet-wifi>sta1 iwconfig
mininet-wifi>sta1 ping sta2 #here I suggest you wait sta1 reaches ap2
    ↪ before going to the next step
mininet-wifi>sta1 iwconfig
```

### 6.2.5 Case 4 - Changes at runtime

```
sudo python examples/position.py
mininet-wifi>sta1 iwconfig
mininet-wifi>sta1 ping sta2
mininet-wifi>py sta1.setPosition('70,40,0')
mininet-wifi>sta1 iwconfig
mininet-wifi>sta1 ping sta2
mininet-wifi>py ap1.setRange(60)
mininet-wifi>sta1 iwconfig
mininet-wifi>sta1 ping sta2
```

### 6.2.6 Case 5 - Bridging physical and virtual emulated environments

```
sudo systemctl stop network-manager
```

Edit sbrc.py:

```
from: phyap1 = net.addPhysicalBaseStation( 'phyap1', ssid=
    ↪ 'SBRC16-MininetWiFi', mode='g', channel='1', position='50,115,0',
    ↪ wlan='wlan11' )
to: wlan11 to your usb wlan interface.
```

Then,

```
sudo python sbrc.py
```

At this moment users attending the conference will be invited to connect their mobile devices into the physical/emulated environment.

## 6.3 SIGCOMM 2016

*Demo: Mininet-WiFi: A Platform for Hybrid Physical-Virtual Software-Defined Wireless Networking Research*

Requirements to reproduce:

- WiFi interface + (other WiFi or ethernet interface)
- Floodlight OpenFlow controller
- ofsoftswitch13 (you may install it with *util/install.sh -3f*) - <https://github.com/CPqD/ofsoftswitch13>
- Speedtest-cli
- Codes are available at <https://github.com/ramonfontes/reproducible-research/tree/master/mininet-wifi/SIGCOMM-2016>

Important (changes in code - You have to set both Internet and wlan interfaces):

```
internetIface = 'eth0' # wired/wireless card.
usbDongleIface = 'wlan0' # wifi interface.
```

Next, executing the Floodlight OpenFlow controller:

```
sudo java -jar target/floodlight.jar
```

Then,

```
sudo py hybridVirtualPhysical.py
mininet-wifi> sh ./rule.hybridVirtualPhysical
```

Despite the content of *rule.hybridVirtualPhysical* is included in *hybridVirtualPhysical.py*, we have faced some troubles in floodlight controller. Thus, probably you have to execute *rule.hybridVirtualPhysical* after executing *hybridVirtualPhysical.py*.

Now, stations should be able to communicate with each other and with the Internet. You may use any station connected to any Access Point and try it out:

```
mininet-wifi>xterm $station
$station>speedtest-cli
```

Using speedtest-cli you can test both Download and Upload speed of your Internet connection. The available bandwidth is controlled by OpenFlow meter entries.

There is a web server accessible at 10.0.0.111 and according rules applied in rule.hybridVirtualPhysical, if you access 10.0.0.109 the traffic will be redirect to 10.0.0.111.

#### Useful commands:

```
sta1 iw dev sta1-wlan0 mpath dump #verify mesh routing information
sh dpctl unix:/tmp/ap3 stats-flow
sh dpctl unix:/tmp/ap3 stats-meter
sh dpctl unix:/tmp/ap3 meter-config
```

### 6.4 From Theory to Experimental Evaluation: Resource Management in Software-Defined Vehicular Networks 2017

Firstly, you have to get vanet.py at <https://github.com/ramonfontes/reproducible-research/blob/master/mininet-wifi/IEEE-Access-2017/vanet.py>

```
sudo python vanet.py
```

Please consider to watch: [https://www.youtube.com/watch?v=kO3O9EwrP\\_s](https://www.youtube.com/watch?v=kO3O9EwrP_s)

### 6.5 The Computer Journal - How far can we go? Towards Realistic Software-Defined Wireless Networking Experiments 2017

Codes necessary to reproduce this paper are available at <https://github.com/ramonfontes/reproducible-research/tree/master/mininet-wifi/The-Computer-Journal-2017>

#### 6.5.1 Wireless n-Casting

##### Requirements:

Floodlight controller.

In order to reproduce this case, please follow the instructions below:

First, run the controller:

```
sudo java -jar target/floodlight.jar
```

start the topology:

```
sudo python ncasting.py
```

and finally install the rules:

```
sudo python ncasting-controller.py
```



### 6.5.2 *Multipath TCP*

#### **Requirements:**

You have to install mptcp and ifstat to reproduce this use case.

In order to allow the communication we use pox controller with spanning tree enabled. This command can be used to enable spanning tree:

```
./pox.py forwarding.l2_learning openflow.spanning_tree --hold-down log.level  
↪ --DEBUG samples.pretty_log openflow.discovery host_tracker  
↪ info.packet_dump
```

Then, start the environment in a new terminal and run some commands with xterm:

```
sudo python mptcp.py  
mininet-wifi>xterm sta1 sta1 h10 h10  
Node: h10 (terminal1)$ifstat  
Node: sta1 (terminal1)$ifstat  
Node: h10 (terminal2)$iperf -s  
Node: sta1 (terminal2)$iperf -c 192.168.1.254
```

### 6.5.3 *Hybrid Physical-Virtual Environment*

See Section 6.3.

### 6.5.4 *SSID-based Flow Abstraction*

#### **Requirements:**

ofsoftswitch13

In order reproduce this case you have to run the following code:

```
sudo python forwardingBySSID.py
```

Then, you may run any application (e.g., Iperf) to test the available bandwidth for any SSID. Alternatively, you might run dpctl to verify meter table configuration.

```
mininet-wifi> sh dpctl unix:/tmp/ap1 meter-config
```

### 6.5.5 *EXPERIMENTAL VALIDATION: Propagation Model*

Consider to use the file propagationModelCase.py if you want to reproduce the results.

### 6.5.6 *EXPERIMENTAL VALIDATION: Simple File Transfer*

Consider to use the files fileTransferring.py and fileTransferring.cc if you want to reproduce the results.

### 6.5.7 *EXPERIMENTAL VALIDATION: Replaying Network Conditions*

Consider to use files into the directory replayingNetwork/ if you want to reproduce the results.

### 6.5.8 On the *Krack Attack*: Reproducing Vulnerability and a Software-Defined Mitigation Approach WCNC 2018

Please refer to <https://github.com/ramonfontes/reproducible-research/tree/master/mininet-wifi/WCNC-2018>

\*





# Mininet-WiFi

Emulator for Software-Defined Wireless Networks

## 7. Publications

### 7.1 SDN For Wireless 2015

*Exhibit*

Fontes, R. R., Afzal, S., Brito, S. H. B., Santos, M., Rothenberg, C. E. “Towards an Emulator for Software Defined Wireless Networks“. In EAI International Conference on Software Defined Wireless Networks and Cognitive Technologies for IoT. Rome, Italy, Oct 2015.

### 7.2 CNSM 2015 - Best Paper Award!

*This paper explains the basic Mininet-WiFi design and useful Case Studies.*

Fontes, R. R., Afzal, S., Brito, S. H. B., Santos, M., Rothenberg, C. E. “Mininet-WiFi: Emulating Software-Defined Wireless Networks“. In 2nd International Workshop on Management of SDN and NFV Systems 2015. Barcelona, Spain, Nov 2015.

### 7.3 SwitchOn 2015

*This paper presents a demo use case in a mobile video streaming scenario to showcase the ability of Mininet-WiFi to emulate the wireless channel in terms of bandwidth, packet loss, and delay variations as a function of the distance between the communicating parties.*

Fontes, R. R., Rothenberg, C. E. Towards an Emulator for Software-Defined Wireless Networks. In: SwitchOn 2015, São Paulo – SP – Brazil.

### 7.4 SBRC 2016

*DEMO*

Ramon dos Reis Fontes and Christian Esteve Rothenberg. Mininet-WiFi: Emulação de Redes Sem Fio Definidas por Software com suporte a Mobilidade. In Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuídos (SBRC 2016) - Salão de Ferramentas, 2016, Salvador - BA - Brazil.

## 7.5 SIGCOMM 2016

### *DEMO*

Ramon dos Reis Fontes and Christian Esteve Rothenberg. Mininet-WiFi: A Platform for Hybrid Physical-Virtual Software-Defined Wireless Networking Research (SIGCOMM 2016) - 2016, Florianopolis - ES - Brazil.

## 7.6 Institute of Electrical and Electronics Engineers - IEEE 2017

*In this paper, we enumerate the potentials of software-defined vehicular networks, analyse the need of rethinking the traditional SDN approach from theoretical and practical standpoints when applied in this application context, and present an emulation approach based on the proposed node car architecture in Mininet- WiFi to showcase the applicability and some expected benefits of SDN in a selected use case scenario.*

Ramon dos Reis Fontes and Claudia Campolo and Christian E. Rothenberg and Antonella Molinaro. From Theory to Experimental Evaluation: Resource Management in Software-Defined Vehicular Networks (IEEE Access 2017), DOI: 10.1109/access.2017.2671030.

## 7.7 The Computer Journal 2017

Ramon dos Reis Fontes and Mohamed Mahfoudi and Walid Dabbous and Thierry Turetletti and Christian Rothenberg. How Far Can We Go? Towards Realistic Software-Defined Wireless Networking Experiments, Oxford University Press (OUP), DOI: 10.1093/comjnl/bxx023.

## 7.8 WCNC 2018

Ramon dos Reis Fontes and Christian Esteve Rothenberg. On the *Krack Attack*: Reproducing Vulnerability and a Software-Defined Mitigation Approach - Poster Session. WCNC 2018.



# Mininet-WiFi

Emulator for Software-Defined Wireless Networks

## 8. Acknowledgment

- This project is partially supported by **INRIA - Institut National de Recherche en Informatique et en Automatique**, in Sophia Antipolis, France and **FAPESP - Fundação de Amparo à Pesquisa do Estado de São Paulo**, in Sao Paulo, Brazil.
- We thank **Dr. Chih-Heng Ke**, from Department of Computer Science and Information Engineering, National Quemoy University, Kinmen, Taiwan, for all our discussion at the beginning of this work.
- We thank **Brian Linkletter** - [brianlinkletter.com](http://brianlinkletter.com) for the tutorial presented at the chapter 4.
- We thank **Patrick Große**, member of <http://www.uni-muenster.de/Comsys/en/> for integrating Mininet-WiFi with wmediumd (section 1.5.2).
- We thank **all users who participate of our mailing** list for collaborating in the development of this work.







# Mininet-WiFi

Emulator for Software-Defined Wireless Networks

## 9. Appendix

### 9.1 handover.py

```
1  #!/usr/bin/python
2
3  'Example for handover'
4
5  from mininet.node import Controller, OVSKernelSwitch
6  from mininet.log import setLogLevel, info
7  from mn_wifi.net import Mininet_wifi
8  from mn_wifi.node import OVSKernelAP
9  from mn_wifi.cli import CLI_wifi
10
11
12  def topology():
13
14      "Create a network."
15      net = Mininet_wifi(controller=Controller, switch=OVSKernelSwitch,
16                          accessPoint=OVSKernelAP)
17
18      info("*** Creating nodes\n")
19      sta1 = net.addStation('sta1', mac='00:00:00:00:00:01', ip='10.0.0.1/8')
20      ap1 = net.addAccessPoint('ap1', ssid='new-ssid1', mode='g', channel='1',
21                              position='15,30,0')
22      ap2 = net.addAccessPoint('ap2', ssid='new-ssid1', mode='g', channel='6',
23                              position='55,30,0')
24      s3 = net.addSwitch('s3')
25      h1 = net.addHost('h1', mac='00:00:00:00:00:02', ip='10.0.0.2/8')
26      c1 = net.addController('c1', controller=Controller, port=6653)
27
28      info("*** Configuring WiFi Nodes\n")
29      net.configureWifiNodes()
30
31      h1.plot(position='35,90,0')
32      s3.plot(position='35,80,0')
```

```
30
31     info("*** Creating links\n")
32     net.addLink(ap1, s3)
33     net.addLink(ap2, s3)
34     net.addLink(h1, s3)
35
36     net.plotGraph(max_x=100, max_y=100)
37
38     net.startMobility(time=0)
39     net.mobility(sta1, 'start', time=1, position='10,30,0')
40     net.mobility(sta1, 'stop', time=80, position='60,30,0')
41     net.stopMobility(time=80)
42
43     info("*** Starting network\n")
44     net.build()
45     cl.start()
46     ap1.start([cl])
47     ap2.start([cl])
48     s3.start([cl])
49
50     info("*** Running CLI\n")
51     CLI_wifi(net)
52
53     info("*** Stopping network\n")
54     net.stop()
55
56 if __name__ == '__main__':
57     setLogLevel('info')
58     topology()
```

Code 9.1: handover.py