Python WiFi API documentation

Jump to bottom

Roberto Riggio edited this page on Aug 18, 2019 · 1 revision

Table of Contents

Overview

In this section we shall first introduce some fundamental concepts like *WTPs*, *LVAPs* and *Resource Blocks*, then we shall describe all the Wi-Fi primitives available to applications developers. Notice how all the methods described in the section are class methods of the `EmpowerApp` class and as such can be invoked only from applications extending this base class.

Wireless Termination Points

Are the a Wi-Fi access points connected to an instance of the EmPOWER Runtime.

The list of WTPs available to the application can be accessed using:

```
self.wtps()
```

Light Virtual Access Point Abstraction

The LVAP abstraction provides a high–level interface for wireless clients state management. A client attempting to join the network, will trigger the creation of a new LVAP. Specifically, in WiFi, an LVAP is a per–client virtual access point which simplifies network management and introduces seamless mobility support, i.e., the LVAP abstraction captures the complexities of the IEEE 802.11 protocol stack such as handling of client associations, authentication, and handovers.

More specifically, every newly received probe request frame from a client at a WTP is forwarded to the controller which may trigger the generation of a probe response frame through the creation of an LVAP at the requesting WTP. The LVAP will thus become a potential candidate AP for the client to perform an association.

Similarly each WTP will host as many LVAPs as the number of wireless clients that are currently under its control. Such LVAP has a BSSID that is specific to the newly associated client. Removing a LVAP from a WTP and instantiating it on another WTP effectively results in a handover.

The list of LVAPs available to the application can be accessed using:

```
self.lvaps()
```

Handovers can be performed in different ways. The easiest is to assign a new WTP instance to the `wtp` property of an LVAP instance:

```
lvap.wtp = new_wtp
```

Resource Block Abstraction

The *Resource Block* is essentially a Wi-Fi interface at a given WTP. A *Resource Block* is identified by three things: the WTP at which the block is available, the channel on which the block is operating, and the type of channel. At the moment there can be two types of channels: Legacy (11b/g/a), and HT20 (11n with 20 MHz channel).

Each WTP instance has a `supports` property which contains the list of resource blocks (i.e. Wi-Fi Channels) supported by the WTP. For example:

```
for wtp in self.wtps():
    print(wtp.supports)
```

This will print the block supported by each WTP available to the *Network App*:

```
{(00:0D:B9:2F:55:CC, 04:F0:21:09:F9:8F, 1, HT20), (00:0D:B9:2F:55:CC, D4:CA:6D:14:C2:0A, 6
```

A shortcut to get the list of all *Resource blocks* available to an applications can be accessed using the following method:

```
blocks = self.blocks()
```

Resource blocks can be filered using different criteria. For example it is possible to filter by channel:

```
self.blocks().filter_by_channel(6)
```

This returns all the resource blocks tuned on channel 6. Similarly all the blocks supporting a certain band can be obtained with (0=Legacy, 1=HT20):

```
self.blocks().filter_by_band(1)
```

The filering operations can also be combined:

```
self.blocks().filter_by_channel(6) \
             .filter_by_band(1)
```

Blocks can also be sorted by rssi with:

```
self.blocks().sort_by_rssi(lvap.addr).first()
```

The code above returns the blocks which are within range of the specified client sorted from the one with better RSSI to the worst. Finally, the *first* returns the first blocks in the list.

The primitives described above can be used to implement a simple handover routine (in the EmpowerApp's loop method):

```
def loop(self):
    for lvap in self.lvaps():
        lvap.blocks = self.blocks().sort_by_rssi(lvap.addr).first()
```

Events

The following table lists the EmPOWER Wi-Fi events.

Primitive	Parameters	Description
lvap_join	lvap	Callback when a LVAP joins a tenant.
lvap_leave	lvap	Callback when a LVAP leaves a tenant.
wtp_up	wtp	Callback when a WTP comes online.
wtp_down	wtp	Callback when a WTP goes offline.

Give a look at this Network App for some examples about how to use events.

Primitives

The following table lists the EmPOWER Wi-Fi primitives. All primitives are non–blocking and, as such, they immediately return control to the calling Network App.

An optional callback method can be specified for Poller/Trigger primitives. The specified callback method will be called when the primitive returns.

All polling primitives are executed periodically with the period set by the *every* parameter (in ms). Specifying `every = -1` will result in a single query being issued.

Primitive	Parameters	Mode	Description
bin_counter	lvap, bins, every	Polling	Aggregate TX/RX packets/bytes in the specified bins for the specified destination address.
ucqm/ncqm	block, every	Polling	Fetch RSSI statistics for neighbouring Stations/APs.
lvap_stats	lvap, every	Polling	Fetch per–link packet transmission statistics.
rssi	addr, relation, value, period	Trigger	Callback when a condition on the RSSI is verified.
slice_stats	dscp, block, bins, every	Polling	Fetch per–slice packet transmission statistics.
summary	addr, block, limit, period	Trigger	Get fine grained link–layer events.
txp_bin_counter	mcast, block, bins, every	Polling	Aggregate TX/RX packets/bytes in the specified bins for the specified multicast address.
wifi_stats	block, every	Polling	Fetch channel occupancy ratio .

Bin Counter

The bin_counter primitive allow programmers to track the traffic exchanged by a certain LVAP and to use binning in order to aggregate such information by frame length (useful in wireless networks due to the fact that short packets incur higher transmission overheads). For example:

```
self.bin_counter(lvap="22:13:a4:cc:de:cb",
                 bins=[512, 1514, 8192],
                 every=5000,
                 callback=self.counters_callback)
```

The statement above instructs the controller to track the packets transmitted and received by a certain LVAP and to aggregate the information into the specified bins. Notice that the query is executed only if the LVAP is associated to the specified SSID otherwise the packet counters are not updated.

Give a look at this Network App for an example about how this use this primitive.

User/Network Channel Quality Maps

The User/Network Channel Quality Map primitive allows the programmer to query a WTP requesting the list of its neighbouring stations or networks in the specified block.

```
self.ucqm(block=block,
          every=5000,
          callback=ucqm_callback)
```

Give a look at this Network App for an example about how this use this primitive.

LVAP Stats

Access to the downlink transmission statistics is possible using the lvap_stats primitive. For each supported MCS the average delivery probability and the expected throughput are reported together with the total number of successful and failed transmissions since the LVAP was moved to the WTP.

```
self.lvap_stats(lvap="22:13:a4:cc:de:cb",,
                every=5000,
                callback=lvap_stats_callback)
```

Give a look at this Network App for an example about how this use this primitive.

RSSI Trigger

The RSSI primitive allows programmers to trigger a callback when the RSSI of an LVAP goes below a certain value. For example:

```
self.rssi(lvap='11:22:33:44:55:66',
          value=-30,
          relation='LT',
          callback=low_rssi)
```

After the trigger has fired for the first time and as long as the RSSI remains below the specified value, the callback method is not called again by the same WTP, however the same callback may be triggered by other WTPs.

Slice State

The slice_stats primitive allow programmers fetch the statistics associated with a certain slice.

```
for block in wtp.supports:
    self.slice_stats(block=block,
                     dscp="0x0",
                     callback=self.slice_stats_callback)
```

The statement above instructs the controller to track the packets transmitted using the dscp *0x0*.

Give a look at this Network App for an example about how this use this primitive.

Link Summary

The summary primitive allows programmers to access link-layer events and the associated meta-data.

```
self.summary(addr='11:22:33:44:55:66',
             block=block,
             limit=10,
             period=2000,
             callback=summary_callback)
```

The statement above generates a periodic callback when a traffic trace has been received from a WTP. The traffic traces includes all the link–layer events within the decoding range of the WTP. The limit parameters instructs the WTP to send only a specified number of traces after which the operation is stopped and all the allocated data structures on the WTP are freed. Specifying −1 results in the WTP sending traffic traces forever. It is also possible to filter link–layer by source MAC Address.

Example of how to handle the callback:

```
def summary_callback(summary):
    self.log.info("New summary from %s addr %s frames %u",
```

```
                summary.block,
                summary.addr,
                len(summary.frames))
```

The frames property contains the meta-data associated to the collected link-layer events:

- Transmitter Address. The MAC address of the transmitter.
- TSFT. The 802.11 MAC's 64-bit Time Synchronization Function Timer. Each frame received by the radio interface is timestamped with a 1µsec resolution clock by the 802.11 driver.
- The frame RSSI (in dB), Rate (in Mb/s), Length (in bytes), and Duration (in µsec).
- Sequence, The 802.11 MAC's 16-bit sequence number. This counter is incremented by the transmitter after a successful transmission.

Give a look at this Network App for an example about how this use this primitive.

TXP Bin Counter

The txp_bin_counter primitive allow programmers fetch the statistics associated with a certain transmission policy. For example lets assume that you want to use 24 Mb/s as MCS for broadcast frames over all the interfaces of a given WTP. This can be done with the following code:

```
for block in wtp.supports:
    tx_policy = block.tx_policies[EtherAddress("ff:ff:ff:ff:ff:ff")]
    tx_policy.mcast = TX_MCAST_LEGACY
    tx_policy.mcs = [24]
```

Now if you want to track the frames transmitted to that address you can use the following code:

```
for block in wtp.supports:
    self.txp_bin_counter(block=block,
                         mcast="ff:ff:ff:ff:ff:ff",
                         callback=self.txp_bin_counter_callback)
```

The statement above instructs the controller to track the packets transmitted to that address by the specified WTP. Notice that despite its name, the *mcast* field can be used also to specify unicast addresses.

Give a look at this Network App for an example about how this use this primitive.

WiFi Stats

The Busyness primitive allows the programmer to query a WTP requesting the channel busyness ratio for the specified block.

```
self.wifi_stats(block=block,
                every=5000,
                callback=busyness_callback)
```

Given a look at this Network App for an example about how this use this primitive.

Support

- Mailing List

Acknowledgements

- Acknowledgements

Clone this wiki locally

```
https://github.com/clicknf/clicknf.github.io.wiki.git
```