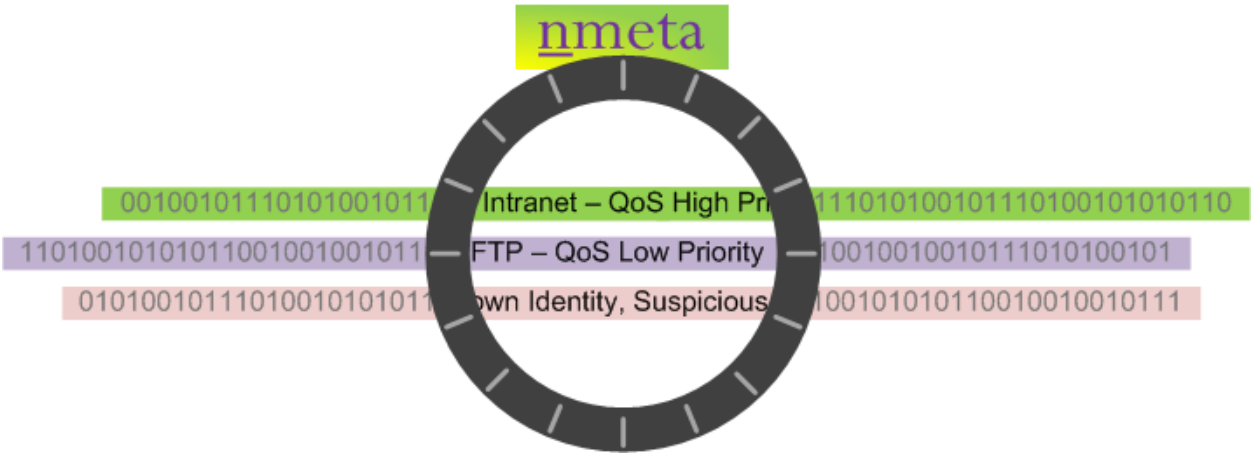
nmeta Documentation

Release 0.2.0

Matthew John Hayes

November 29, 2016

Contents



The nmeta project is a research platform for traffic classification on Software Defined Networking (SDN). Read More

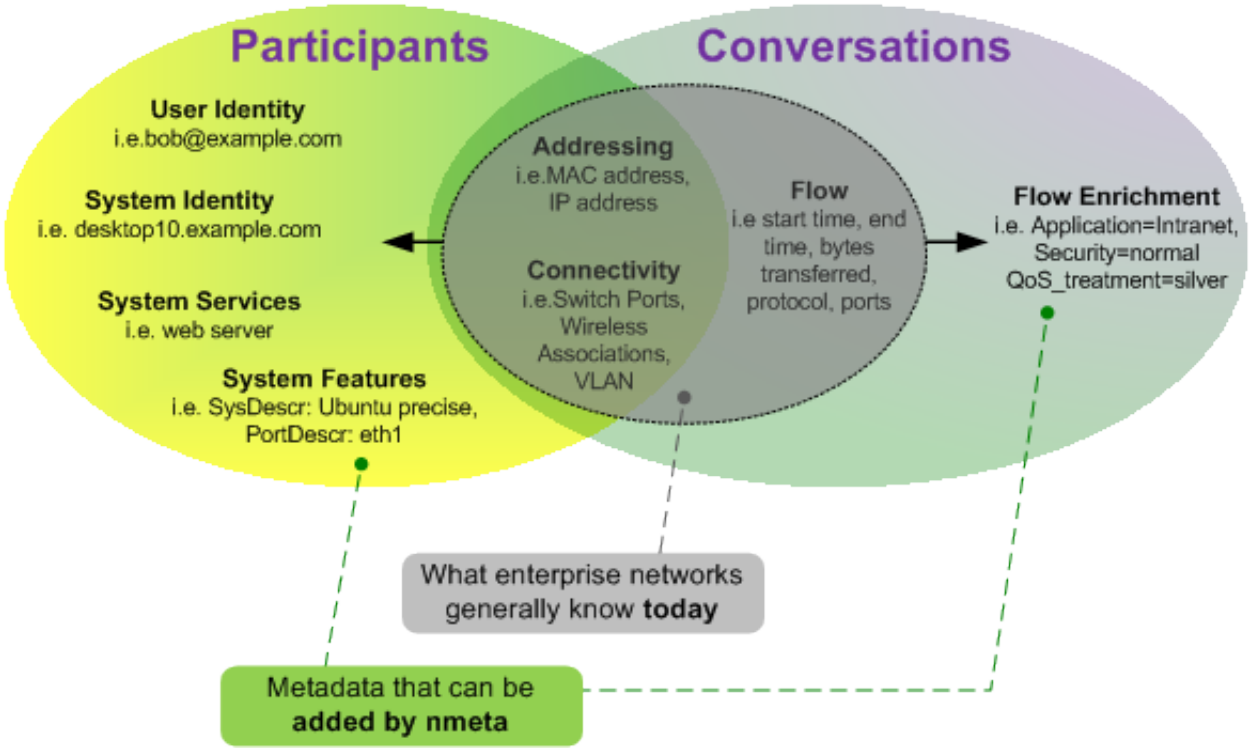
Contents:

CHAPTER 1

Introduction

The nmeta *(pronounced en-meta)* project is founded on the belief that innovation in networks requires a foundation layer of knowledge about both the participants and their types of conversation.

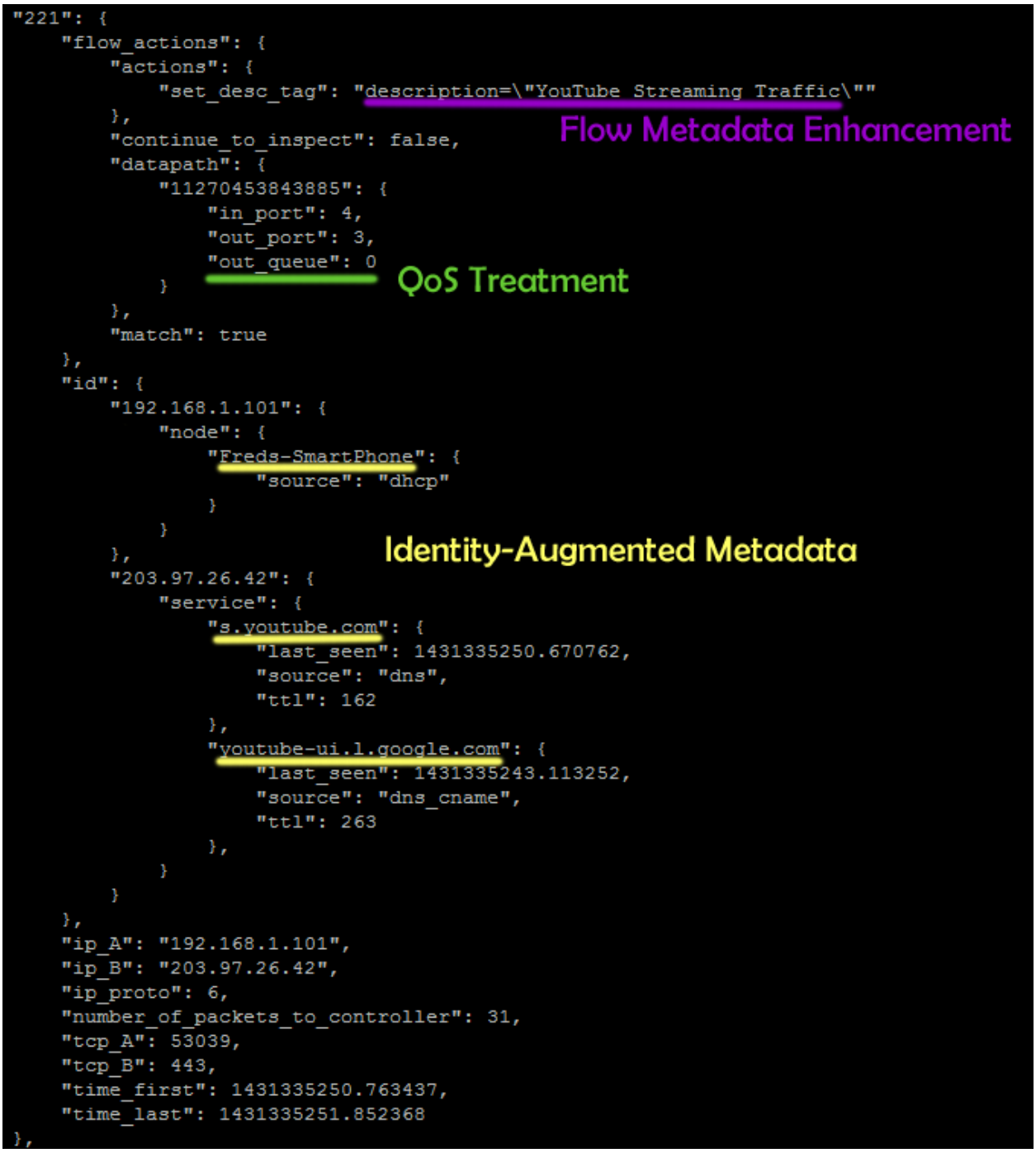
Today, networks generally have only a limited view of participants and conversation types



The goal of the nmeta project is to produce network metadata enriched with participant identities and conversation types to provide a foundation for innovation in networking.

The production of enriched network metadata requires policy-based control, and ability to adapt to new purposes through extensibility.

Enriched network metadata has a number of uses, including classifying flows for QoS, billing, traffic engineering, troubleshooting and security.



Nmeta is a research platform for traffic classification on Software Defined Networking (SDN). It runs on top of the Ryu SDN controller (see: http://osrg.github.io/ryu/).

1.1 Limitations

Nmeta does not scale well. Every new flow has a processing overhead, and this workload cannot be scaled horizontally on the controller. The nmeta2 system is being developed to address this limitation.

1.2 Feature Enhancement Wishlist

See Issues for list of enhancements and bugs

1.3 Privacy Considerations

Collecting network metadata brings with it ethical and legal considerations around privacy. Please ensure that you have permission to monitor traffic before deploying this software.

1.4 Disclaimer

This code carries no warrantee whatsoever. Use at your own risk.

1.5 How to Contribute

Code contributions and suggestions are welcome. Enhancement or bug fixes can be raised as issues through GitHub.

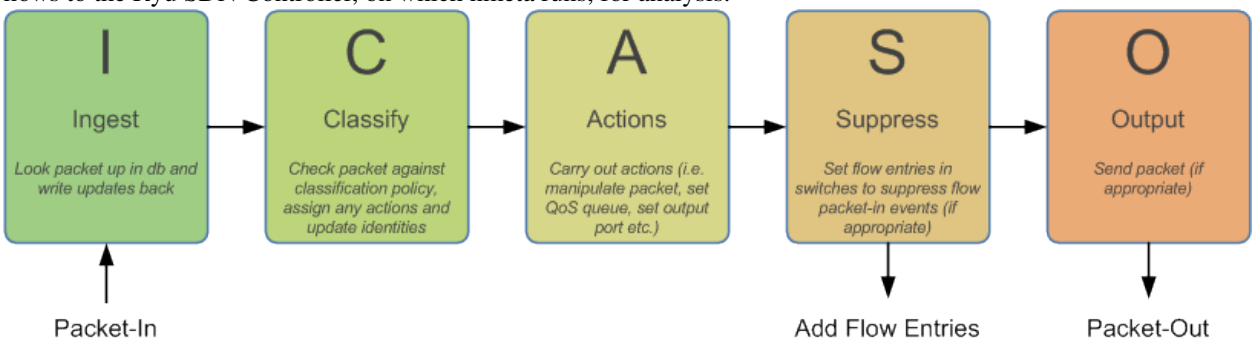
Please get in touch if you want to be added as a contributor to the project:

Email: Nmeta Maintainer

CHAPTER 2

How it Works

Nmeta uses OpenFlow Software-Defined Networking (SDN) to selectively control flows through switches so that packets can be classified and actions taken. It instructs connected OpenFlow switches to send packets from unknown flows to the Ryu SDN Controller, on which nmeta runs, for analysis.



Nmeta configures a single flow table per switch with a table-miss flow entry (FE) that sends full unmatched packets to the controller. As flows are classified, specific higher-priority FEs are configured to suppress sending further packets to the controller.

CHAPTER 3

Install

This guide is for installing on Ubuntu.

3.1 Pre-Work

3.1.1 Ensure packages are up-to-date

```
sudo apt-get update
sudo apt-get upgrade
```

3.1.2 Install Python pip

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

3.1.3 Install git

Install git and git-flow for software version control:

```
sudo apt-get install git git-flow
```

3.2 Install Ryu OpenFlow Controller

Ryu is the OpenFlow Software-Defined Networking (SDN) controller application that handles communications with the switch:

```
sudo pip install ryu
```

3.3 Install Packages Required by nmeta

3.3.1 Install dpkt library

The dpkt library is used to parse and build packets:

```
sudo pip install dpkt
```

3.3.2 Install pytest

Pytest is used to run unit tests:

```
sudo apt-get install python-pytest
```

3.3.3 Install YAML

Install Python YAML ("YAML Ain't Markup Language") for parsing config and policy files:

```
sudo apt-get install python-yaml
```

3.3.4 Install simplejson

```
sudo pip install simplejson
```

3.3.5 Install eve

Eve is used to power the external API

```
pip install eve
```

3.3.6 Install coloredlogs

Install coloredlogs to improve readability of terminal logs by colour-coding:

```
sudo pip install coloredlogs
```

3.3.7 TBD

mongodb + pymongo

3.4 Install nmeta

Clone nmeta

```
cd
git clone https://github.com/mattjhayes/nmeta.git
```

3.5 Run nmeta

```
cd
cd ryu
PYTHONPATH=. ./bin/ryu-manager ../nmeta/nmeta/nmeta.py
```

3.6 Aliases

Aliases can be used to make it easier to run common commands. To add the aliases, edit the .bash_aliases file in your home directory:

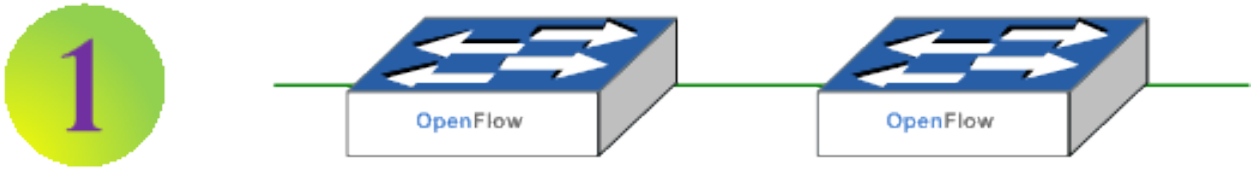
```
cd
sudo vi .bash_aliases
```

Paste in the following:

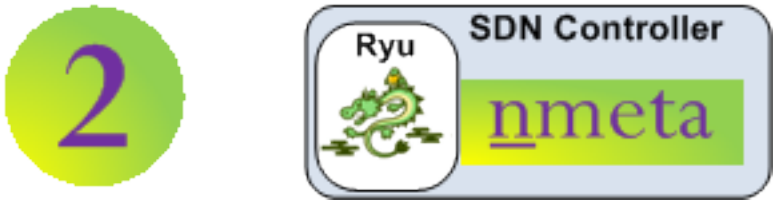
```
# Run nmeta:
alias nm="cd; cd ryu; PYTHONPATH=. ./bin/ryu-manager ../nmeta/nmeta/nmeta.py"
#
# Retrieve nmeta network metadata:
alias idmac="sudo python nmeta/misc/jsonpretty.py http://127.0.0.1:8080/nmeta/identity/mac/"
alias idip="sudo python nmeta/misc/jsonpretty.py http://127.0.0.1:8080/nmeta/identity/ip/"
alias idsvc="sudo python nmeta/misc/jsonpretty.py http://127.0.0.1:8080/nmeta/identity/service/"
alias idsys="sudo python nmeta/misc/jsonpretty.py http://127.0.0.1:8080/nmeta/identity/systemtable/"
alias idnic="sudo python nmeta/misc/jsonpretty.py http://127.0.0.1:8080/nmeta/identity/nictable/"
```

CHAPTER 4

Quick Start Guide



First, you'll need an **OpenFlow Network** with one or more switches. If you don't have a suitable one to hand then consider building the virtual lab in the Extras section



Next, you'll need an **SDN Controller** to run the control plane of the network and host the nmeta application. If you built the virtual lab then you've already got this covered.

If not, build a physical or virtual server. The preferred OS is Ubuntu. Now install Ryu and nmeta as per the Install Guide

You'll need some **participants** (hosts) on your network. Again, if you've built the virtual lab you're already covered for this.

If not, decide what types and numbers of hosts you want on your network, then connect them up.


Configure nmeta as per the Config Guide

Run nmeta:

```
cd
cd ryu
PYTHONPATH=. ./bin/ryu-manager ../nmeta/nmeta.py
```

Now start experimenting. Use the calls in the aliases to show network metadata

CHAPTER 5

Configure Nmeta

5.1 System Configuration

A YAML file holds the system configuration. It's location is:

```
~/nmeta/nmeta/config/config.yaml
```

5.2 Configure Main Policy

The main policy configures how nmeta works with data plane traffic. This includes traffic classification rules. The main policy is stored in the YAML file:

```
~/nmeta/nmeta/config/main_policy.yaml
```

It is used to control what classifiers are used, in what order and what actions are taken.

The traffic classification policy is based off a root key *tc_rules*. This root contains a *ruleset* name (only one ruleset supported at this stage), which in turn contains one or more *rules*.

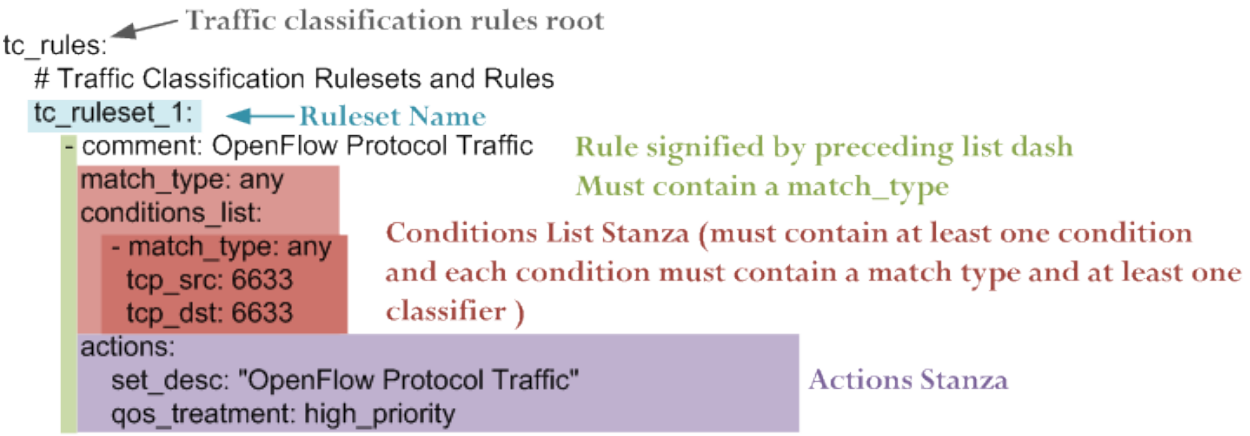
Rules are an ordered list (denoted by preceding dash). Each rule contains:

Comment A *comment* to describe the purpose of the rule (optional). A comment must start with the attribute *comment:* and any single-line string can follow

Match Type A *match type* is one of *any* or *all*

Conditions List A single *conditions_list* stanza that contains one or more *conditions* stanzas

Example simple traffic classification policy with a single rule:



A *conditions_list* stanza contains:

- A match type, consisting of *any* or *all*
- One or more *conditions* as list items (denoted by dash preceding the first item)
- One or more *classifiers* (see below)

A *conditions* stanza is a list item in a conditions list and contains:

- A match type, consisting of *any* or *all*
- One or more *classifiers* (see below)

A *actions* stanza contains one or more attribute/value pairs

Here is a more complex traffic classification policy:

Conditions invoke classifiers. There are three types of classifier supported:

- Static
- Identity
- Custom (Payload / Statistical / Multi-classifier)

5.2.1 Static Classifiers

Static classifiers match on attributes in packet headers, or on environmental attributes such as port numbers.

Supported attributes are:

eth_src Ethernet source MAC address.

Example:

```
eth_src: 08:00:27:4a:2d:41
```

eth_dst Ethernet destination MAC address.

Example:

```
        eth_dst: 08:00:27:4a:2d:42
```

eth_type Ethernet type. Can be in hex (starting with 0x) or decimal.

Examples:

```
        eth_type: 0x0800
```

```
        eth_type: 35020
```

ip_src IP source address. Can be a single address, a network with a mask in CIDR notation, or an IP range with two addresses separated by a hyphen. Both addresses in a range must be the same type, and the second address must be higher than the first.

Examples:

```
        ip_src: 192.168.56.12
```

```
        ip_src: 192.168.56.0/24
```

```
        ip_src: 192.168.56.12-192.168.56.31
```

ip_dst IP destination address. Can be a single address, a network with a mask in CIDR notation, or an IP range with two addresses separated by a hyphen. Both addresses in a range must be the same type, and the second address must be higher than the first.

Examples:

```
        ip_dst: 192.168.57.40
```

```
        ip_dst: 192.168.57.0/24
```

```
        ip_dst: 192.168.57.36-192.168.78.31
```

tcp_src TCP source port.

Example:

```
        tcp_src: 22
```

tcp_dst TCP destination port.

Example:

```
        tcp_dst: 80
```

5.2.2 Identity Classifiers

All identity classifiers are prefixed with:

```
identity_
```

LLDP systemname may be matched as a regular expression. The match pattern must be contained in single quotes. For example, to match system names of *.audit.example.com, add this policy condition:

```
identity_lldp_systemname_re: '.*\.audit\.example\.com'
```

Supported attributes are:

identity_lldp_systemname Exact match against a system name discovered via LLDP. Example:

```
        identity_lldp_systemname: bob.example.com
```

identity_lldp_systemname_re Regular expression match against a system name discovered via LLDP. Example:

```
        identity_lldp_systemname_re: '.*\.audit\.example\.com'
```

identity_service_dns

Exact match of either IP address in a flow against a DNS domain. Example:

```
        identity_service_dns: www.example.com
```

identity_service_dns_re Regular expression match of either IP address in a flow against a DNS domain. Example:

```
        identity_service_dns_re: '.*\.example\.com'
```

5.2.3 Custom Classifiers

Nmeta supports the creation of custom classifiers.

All custom classifiers have the attribute:

```
custom
```

The value determines the custom .py file to load from the nmeta/classifiers directory

For example, the following condition loads a custom classifier file ~/nmeta/nmeta/classifiers/statistical_qos_bandw

```
custom: statistical_qos_bandwidth_1
```

5.2.4 Actions

Actions are specific to a rule, and define what nmeta should do when the rule is matched.

Supported attributes are:

qos_treatment Specify QoS treatment for flow.

Example:

```
        qos_treatment: classifier_return
```

Values can be:

- default_priority
- constrained_bw
- high_priority
- low_priority
- classifier_return

set_desc Set description for the flow. This is a convenience for humans.

Example:

```
        set_desc: "This is a flow type description"
```

5.2.5 QoS Treatment

Quality of Service (QoS) treatment parameters are configured in main policy under the qos_treatment root directive. They map qos action values to queue numbers. Example:

```
qos_treatment:
  # Control Quality of Service (QoS) treatment mapping of
  #  names to output queue numbers:
  default_priority: 0
  constrained_bw: 1
  high_priority: 2
  low_priority: 3
```

CHAPTER 6

API Guide

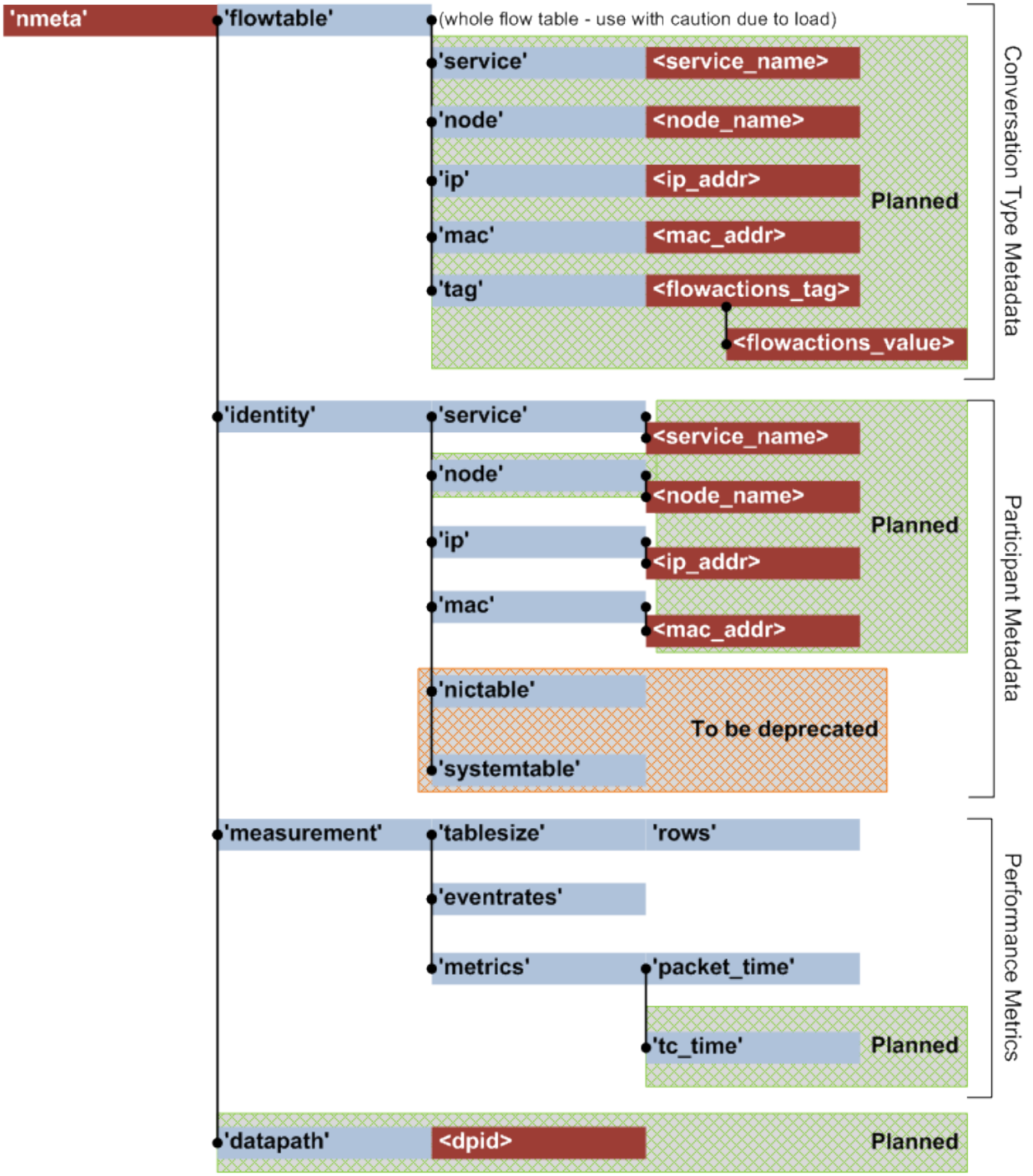
The nmeta API provides HTTP read access (no write at this stage) to data within nmeta. Data includes:

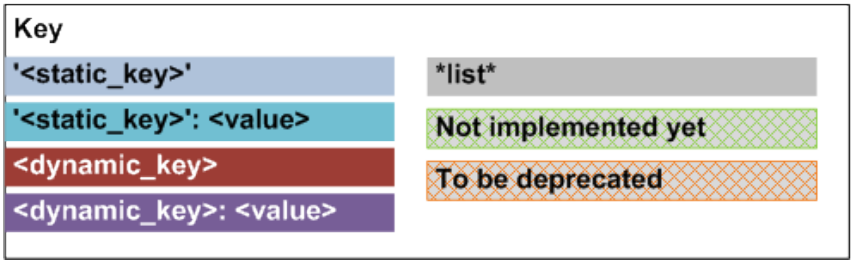
Conversation Type Metadata The types of conversations that are occuring over the network

Participant Metadata Who and what is connected to the network

Performance Metrics How the system is performing

Here is a visualisation of the API hierarchy:


'nmeta'
'flowtable'
(whole flow table - use with caution due to load)
'service'
<service_name>
'node'
<node_name>
'ip'
<ip_addr>
'mac'
<mac_addr>
'tag'
<flowactions_tag>
<flowactions_value>
Planned
Conversation Type Metadata
'identity'
'service'
<service_name>
'node'
<node_name>
'ip'
<ip_addr>
'mac'
<mac_addr>
Planned
'nictable'
'systemtable'
To be deprecated
Participant Metadata
'measurement'
'tablesize'
'rows'
'eventrates'
'metrics'
'packet_time'
'tc_time'
Planned
Performance Metrics
'datapath'
<dpid>
Planned


Key
'<static_key>'
'<static_key>': <value>
<dynamic_key>
<dynamic_key>: <value>
list
Not implemented yet
To be deprecated

To return the JSON in a human-friendly format, precede the API call with the jsonpretty.py script (requires install of simplejson):

```
sudo python ~/nmeta/misc/jsonpretty.py API_CALL_HERE
```

6.1 Example API Calls

Example API Calls to run on local host (jsonpretty.py omitted for brevity):

6.1.1 Conversation Type Metadata

Return the Flow Metadata Table:

```
http://127.0.0.1:8080/nmeta/flowtable/

Returns the whole flow table - use with caution due to load considerations
```

6.1.2 Participant Metadata

Return the Identity MAC structure:

```
http://127.0.0.1:8080/nmeta/identity/mac/
```

Return the Identity IP structure:

```
http://127.0.0.1:8080/nmeta/identity/ip/
```

Return the Identity Service structure:

```
http://127.0.0.1:8080/nmeta/identity/service/
```

Return the Identity NIC Table (old - will be deprecated at some stage):

```
http://127.0.0.1:8080/nmeta/identity/nictable/
```

Return the Identity System Table (old - will be deprecated at some stage):

```
http://127.0.0.1:8080/nmeta/identity/systemtable/
```

6.1.3 Performance Metrics

Return the Flow Metadata table size as number of rows:

```
http://127.0.0.1:8080/nmeta/measurement/tablesize/rows/
```

Return the rate at which nmeta is processing events from switches, as events per second:

```
http://127.0.0.1:8080/nmeta/measurement/eventrates/
```

Return statistics on nmeta per-packet processing time:

```
http://127.0.0.1:8080/nmeta/measurement/metrics/packet_time/
```

CHAPTER 7

Logging

Logging is controlled by the system configuration YAML file:

```
~/nmeta/nmeta/config/config.yaml
```

Logging is separately configured for syslog and to the console, and levels are configurable per Python module. The log format is also customisable.

CHAPTER 8

Data Structures

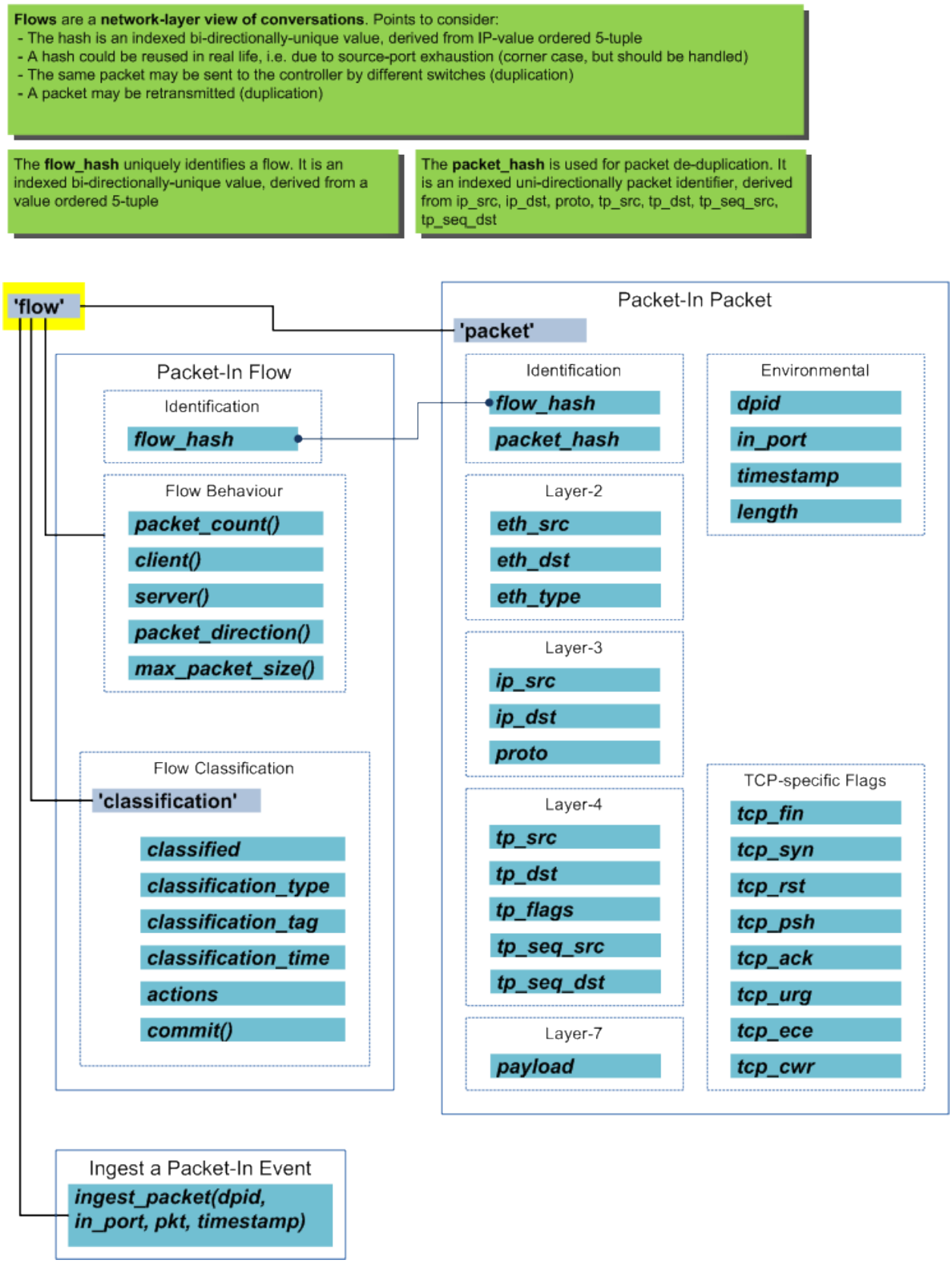
Nmeta uses various data structures to store network metadata related to participants and flows (conversations).

High level abstractions of participants and flows abstract the details of the various MongoDB collections.

8.1 Information Abstractions

8.1.1 Flows Abstraction

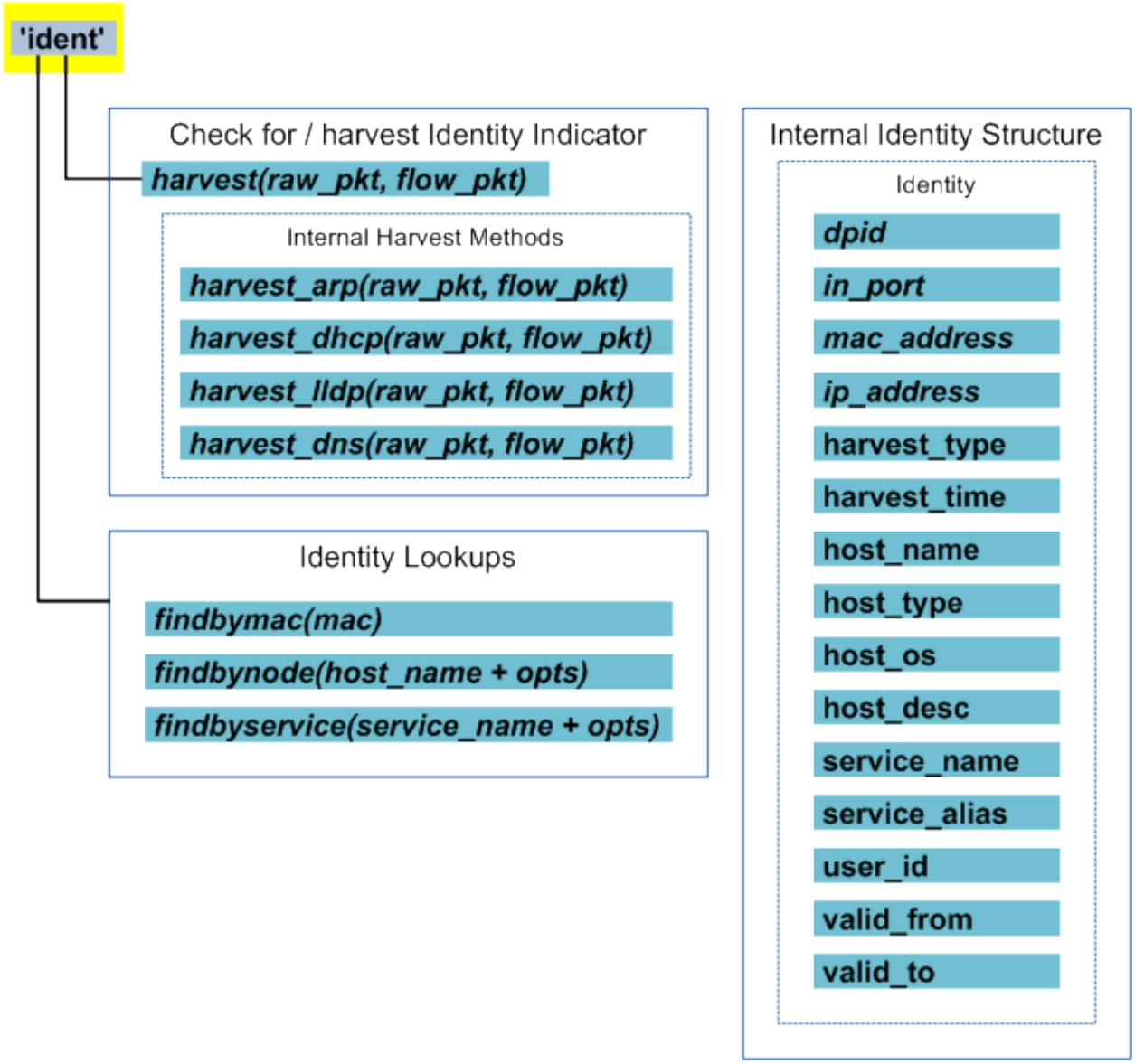
The flows object provides an abstraction of flows (conversations) that have been seen on the network. Flow metrics are in the context of the flow that the last packet-in ingested packet belonged to. The packet context is likewise that of the packet from that event.



Classifiers can make use of the flows object to gain easy access to features of the current flow.

8.1.2 Identities Abstraction

The identities object provides an abstraction for participants (identities) that are known to nmeta. Classifiers can use the identities object to look up the identity information of participants.



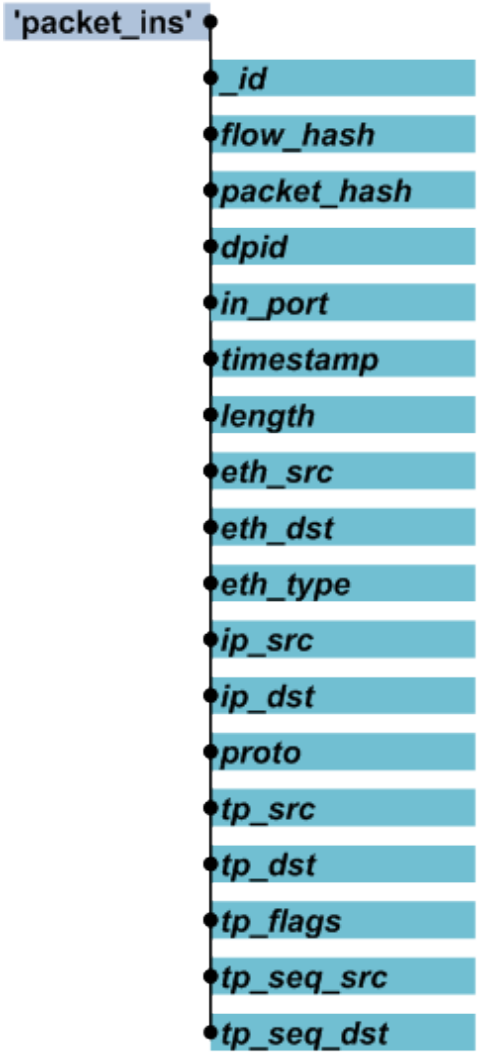
8.2 Database Collections

Nmeta uses capped MongoDB database collections to obviate the need to maintain size by pruning old entries.

8.2.1 Packet-Ins

MongoDB Collection: packet_ins

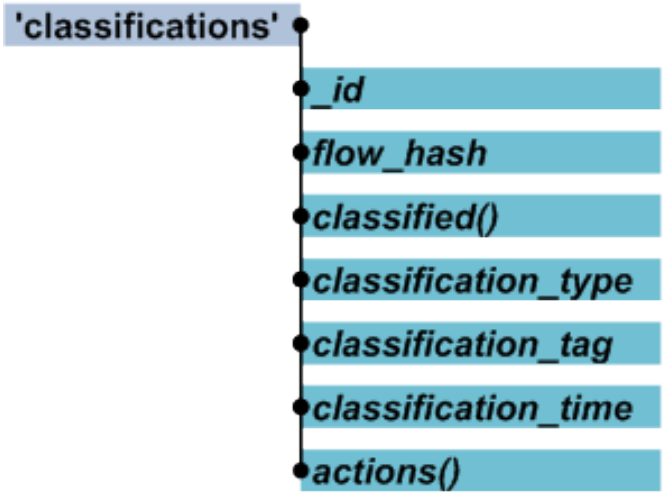
Documents in the **packet_ins** MongoDB collection are **immutable**, so that they can be stored in a **capped collection**, which is fast and automatically overwrites oldest documents when it reaches max size in bytes. Documents in capped collections cannot be increased in size, so it is easiest to treat them as immutable. Their data can be mined for flow metadata, packet data and performance metrics.



8.2.2 Classifications

MongoDB Collection: classifications

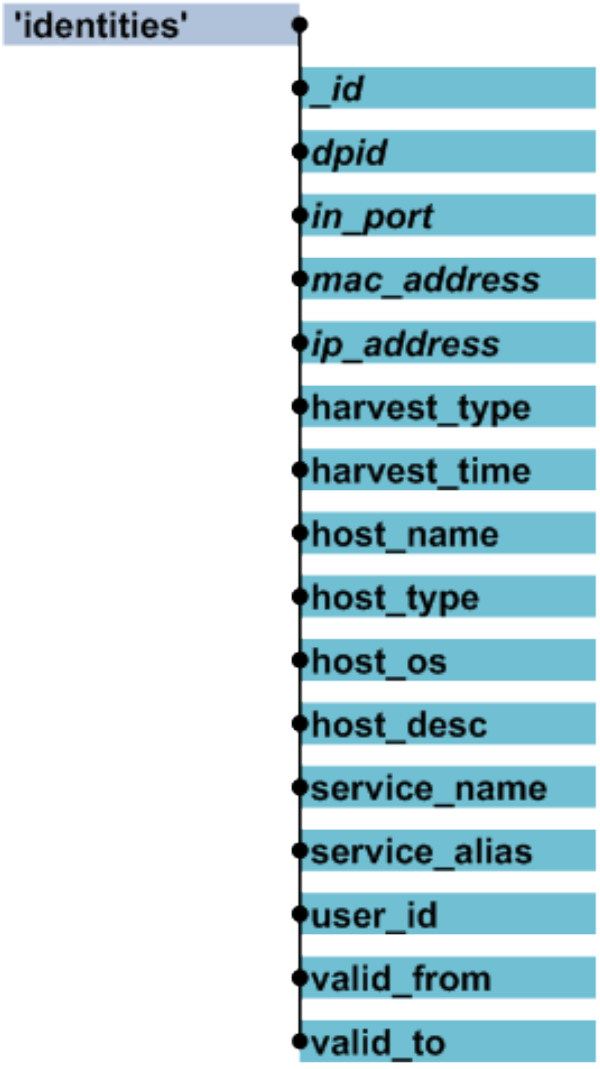
The **classifications** database collection is holds a record of traffic classifications made by nmeta, along with actions to take. Points to consider:
- The *classified* flag says whether or not more packets need to be seen to make a determination



8.2.3 Identity Metadata

MongoDB Collection: identities

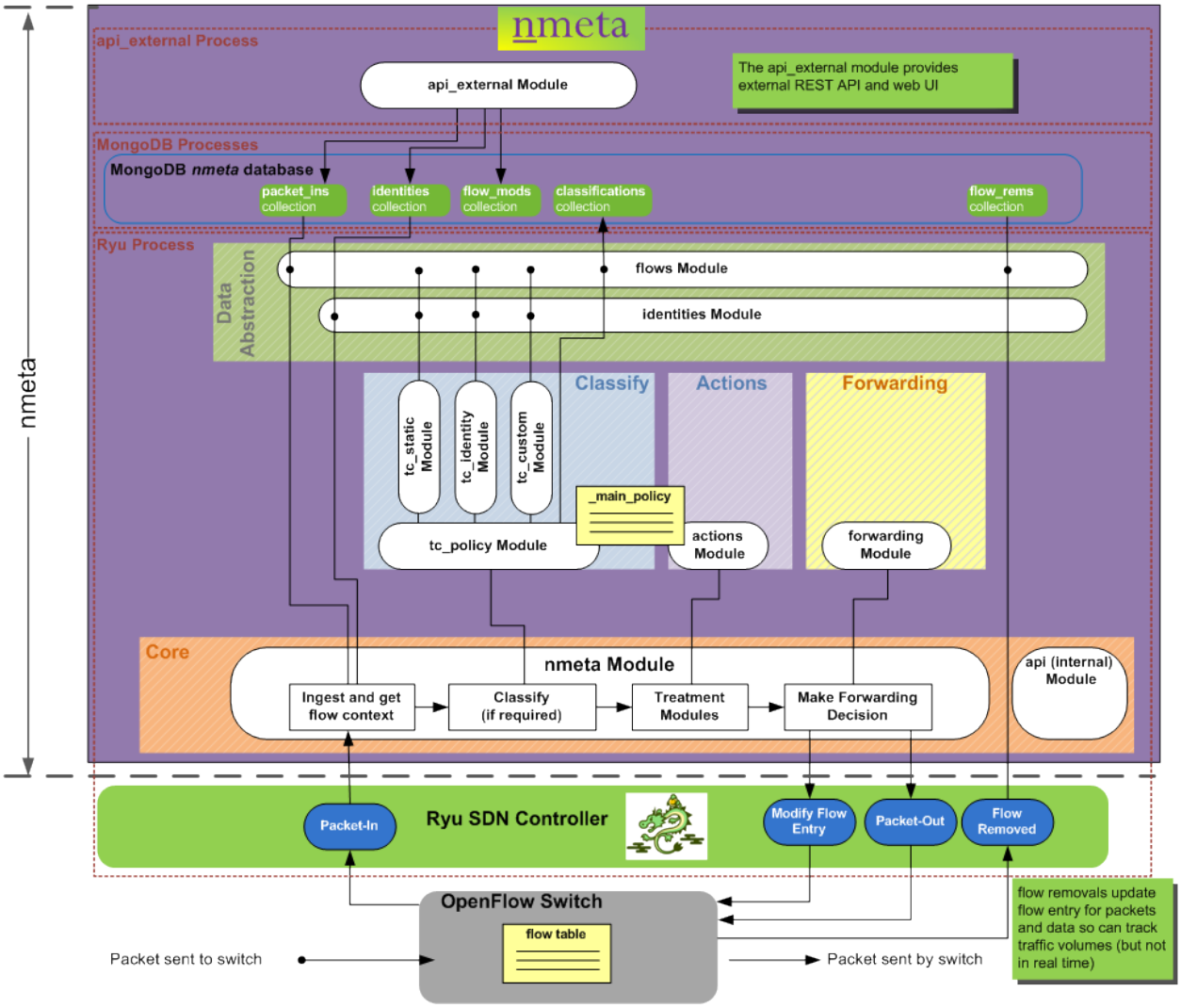
The **identities** database collection is a **view of network participants**. Points to consider:
- Assumed that a participant always has an IP address (beware: this may not always hold true)
- Any valid IP version (i.e. IPv4 or IPv6)
- An IP address can be shared by multiple participants concurrently (examples: participants behind a proxy server or sharing a server)
- An IP address may be dedicated to a participant, but only for a period of time (example: DHCP-assigned address on Wi-Fi)



CHAPTER 9

Code Structure

9.1 High Level



CHAPTER 10

Code Documentation

10.1 nmeta module

This is the main module of the nmeta suite running on top of Ryu SDN controller to provide network identity and flow (traffic classification) metadata . Do not use this code for production deployments - it is proof of concept code and carries no warrantee whatsoever. You have been warned.

class nmeta.**NMeta**(**args*, ***kwargs*)

Bases: ryu.base.app_manager.RyuApp, baseclass.BaseClass

This is the main class used to run nmeta

OFP_VERSIONS = [1, 4]

_CONTEXTS = {'wsgi': <class 'ryu.app.wsgi.WSGIApplication'>}

_add_flow(*ev*, *in_port*, *out_port*, *out_queue*)

Add a flow entry to a switch Prefer to do fine-grained match where possible

_port_status_handler(*ev*)

Switch Port Status event

desc_stats_reply_handler(*ev*)

Receive a reply from a switch to a description statistics request

error_msg_handler(*ev*)

A switch has sent us an error event

flow_removed_handler(*ev*)

A switch has sent an event to us because it has removed a flow from a flow table

packet_in(*ev*)

This method is called for every Packet-In event from a Switch. We receive a copy of the Packet-In event, pass it to the traffic classification area for analysis, work out the forwarding, update flow metadata, then add a flow entry to the switch (when appropriate) to suppress receiving further packets on this flow. Finally, we send the packet out the switch port(s) via a Packet-Out message, with appropriate QoS queue set.

switch_connection_handler(*ev*)

A switch has connected to the SDN controller. We need to do some tasks to set the switch up properly such as setting it's config for fragment handling and table miss packet length and requesting the switch description

nmeta.**ipv4_text_to_int**(*ip_text*)

Takes an IP address string and translates it to an unsigned integer

10.2 tc_policy module

This module is part of the nmeta suite running on top of Ryu SDN controller to provide network identity and flow (Traffic Classification - TC) metadata. It expects a file called "main_policy.yaml" to be in the config subdirectory containing properly formed YAML that conforms the the particular specifications that this program expects. See constant tuples at start of program for valid attributes to use.

class tc_policy.**TrafficClassificationPolicy**(*config*, *pol_dir='config'*, *pol_file='main_policy.yaml'*)

Bases: baseclass.BaseClass

This class is instantiated by nmeta.py and provides methods to ingest the policy file main_policy.yaml and check flows against policy to see if actions exist

class Condition

Bases: object

An object that represents a traffic classification condition, including any decision collateral on match test

to_dict()

Return a dictionary object of the condition object

class TrafficClassificationPolicy.**Conditions**

Bases: object

An object that represents traffic classification conditions, including any decision collateral on matches and actions

to_dict()

Return a dictionary object of the condition object

class TrafficClassificationPolicy.**Rule**

Bases: object

An object that represents a traffic classification rule (a set of conditions), including any decision collateral on matches and actions

to_dict()

Return a dictionary object of the condition object

TrafficClassificationPolicy.**check_policy**(*flow*, *ident*)

Passed a flows object, set in context of current packet-in event, and an identities object. Check if packet matches against any policy rules and if it does, update the classifications portion of the flows object to reflect details of the classification.

TrafficClassificationPolicy.**qos**(*qos_treatment*)

Passed a QoS treatment string and return the relevant QoS queue number to use, otherwise 0. Works by lookup on qos_treatment section of main_policy

TrafficClassificationPolicy.**validate_policy**()

Check main policy to ensure that it is in correct format so that it won't cause unexpected errors during packet checks.

10.3 tc_static module

This module is part of the nmeta suite running on top of Ryu SDN controller to provide network identity and flow (traffic classification) metadata

class tc_static.**StaticInspect**(*config*)
Bases: baseclass.BaseClass

This class is instantiated by tc_policy.py (class: TrafficClassificationPolicy) and provides methods to query static traffic classification matches

check_static(*condition*, *pkt*)
Passed condition and flows packet objects Update the condition match with boolean of result of match checks

is_match_ethertype(*value_to_check1*, *value_to_check2*)
Passed a two prospective EtherTypes and check to see if they are the same. Return 1 for both the same EtherType and 0 for different Values can be hex or decimal and are 2 bytes in length

is_match_ip_space(*ip_addr*, *ip_space*)
Passed an IP address and an IP address space and check if the IP address belongs to the IP address space. If it does return 1 otherwise return 0

is_match_macaddress(*value_to_check1*, *value_to_check2*)
Passed a two prospective MAC addresses and check to see if they are the same address. Return 1 for both the same MAC address and 0 for different

is_valid_ethertype(*value_to_check*)
Passed a prospective EtherType and check that it is valid. Can be hex (0x*) or decimal Return 1 for is valid IP address and 0 for not valid

is_valid_ip_space(*value_to_check*)
Passed a prospective IP address and check that it is valid. Can be IPv4 or IPv6 and can be range or have CIDR mask Return 1 for is valid IP address and 0 for not valid

is_valid_macaddress(*value_to_check*)
Passed a prospective MAC address and check that it is valid. Return 1 for is valid IP address and 0 for not valid

is_valid_transport_port(*value_to_check*)
Passed a prospective TCP or UDP port number and check that it is an integer in the correct range. Return 1 for is valid port number and 0 for not valid port number

10.4 tc_identity module

This module is part of the nmeta suite running on top of Ryu SDN controller to provide network identity and flow (traffic classification) metadata

class tc_identity.**IdentityInspect**(*config*)
Bases: baseclass.BaseClass

This class is instantiated by tc_policy.py (class: TrafficClassificationPolicy) and provides methods to ingest identity updates and query identities

check_dns(*dns_name*, *pkt*, *ident*, *is_regex=False*)
Passed a DNS name, flows packet object, an instance of the identities class and a regex boolean (if true, DNS name is treated as regex). Return True or False based on whether or not the packet has a source or destination IP address that has been resolved from the DNS name. Uses methods of the Identities class to work this out. Returns boolean

check_identity(*condition*, *pkt*, *ident*)
Checks if a given packet matches a given identity match rule. Passed condition, flows packet and identities objects and update the condition match based on whether or not either of the packet IP addresses matches the identity attribute/value. Uses methods of the Identities class to work this out

check_lldp(*host_name*, *pkt*, *ident*, *is_regex=False*)
Passed a hostname, flows packet object, an instance of the identities class and a regex boolean (if true, hostname is treated as regex). Return True or False based on whether or not the packet has a source or destination IP address that matches the IP address registered to the given hostname (if one even exists). Uses methods of the Identities class to work this out. Returns boolean

10.5 tc_custom module

This module is part of the nmeta suite running on top of Ryu SDN controller to provide network identity and flow (traffic classification) metadata

class tc_custom.**CustomInspect**(*config*)
Bases: baseclass.BaseClass

This class is instantiated by tc_policy.py (class: TrafficClassificationPolicy) and provides methods to run custom traffic classification modules

check_custom(*condition*, *flow*, *ident*)
Passed condition, flows and identities objects. Call the named custom classifier with these values so that it can update the condition match as appropriate.

instantiate_classifiers(*custom_list*)
Dynamically import and instantiate classes for any custom classifiers specified in the controller nmeta2 main_policy.yaml

Passed a deduplicated list of custom classifier names (without .py) to load.

Classifier modules live in the 'classifiers' subdirectory

10.6 api module

This module is part of the nmeta suite running on top of Ryu SDN controller to provide network identity and flow metadata. It provides methods for RESTful API connectivity.

class api.**Api**(*_nmeta*, *_config*, *_wsgi*)
Bases: object

This class is instantiated by nmeta.py and provides methods for RESTful API connectivity.

IP_PATTERN = '\\b(25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)(\\.|$){4}\\b'

url_data_size_rows = '/nmeta/measurement/tablesize/rows/'

url_flowtable = '/nmeta/flowtable/'

url_flowtable_augmented = '/nmeta/flowtable/augmented/'

url_flowtable_by_ip = '/nmeta/flowtable/{ip}'

url_identity_ip = '/nmeta/identity/ip/'

url_identity_mac = '/nmeta/identity/mac/'

url_identity_nic_table = '/nmeta/identity/nictable/'

url_identity_service = '/nmeta/identity/service/'

url_identity_system_table = '/nmeta/identity/systemtable/'

url_measure_event_rates = '/nmeta/measurement/eventrates/'

url_measure_pkt_time = '/nmeta/measurement/metrics/packet_time/'

exception api.**NotFoundError**(*msg=None*, ***kwargs*)
Bases: ryu.exception.RyuException

message = 'Error occurred talking to function <TBD>'

class api.**RESTAPIController**(*req*, *link*, *data*, ***config*)
Bases: ryu.app.wsgi.ControllerBase

This class is used to control REST API access to the nmeta data and control functions

get_data_structure_size_rows(**args*, ***kwargs*)
Run a REST command and return appropriate response

get_event_rates(**args*, ***kwargs*)
Run a REST command and return appropriate response

get_id_ip(**args*, ***kwargs*)
Run a REST command and return appropriate response

get_id_mac(**args*, ***kwargs*)
Run a REST command and return appropriate response

get_id_service(**args*, ***kwargs*)
Run a REST command and return appropriate response

get_packet_time(**args*, ***kwargs*)
Run a REST command and return appropriate response

list_flow_table(**args*, ***kwargs*)
Run a REST command and return appropriate response

list_flow_table_augmented(**args*, ***kwargs*)
Run a REST command and return appropriate response

list_flow_table_by_IP(**args*, ***kwargs*)
Run a REST command and return appropriate response

list_identity_nic_table(**args*, ***kwargs*)
Run a REST command and return appropriate response

list_identity_system_table(**args*, ***kwargs*)
Run a REST command and return appropriate response

api.**rest_command**(*func*)
REST API command template

10.7 api_external module

The api_external module is part of the nmeta suite, but is run separately

This module runs a class and methods for an API that exposes an interface into nmeta MongoDB collections.

It leverages the Eve Python REST API Framework

class api_external.**ExternalAPI**(*config*)
Bases: baseclass.BaseClass

This class provides methods for the External API

ingest_dictionary(*filename*)
Read text file that is in dictionary format into a Python dictionary object. Uses ast module.

m_pi_rate_response(*items*)
Update the response with the packet_in rate. Hooked from on_fetched_resource_<name>

on_inserted_callback(*resource_name*, *items*)
Runs on Decision API database inserts, after database insertion completed. It places a message onto the multi-process queue that contains link to resource in database

post_get_callback(*resource*, *request*, *payload*)
TBD

pre_get_callback(*resource*, *request*, *lookup*)
Runs on GET request pre database lookup

run()
Run the External API instance

10.8 config module

This module is part of the nmeta suite running on top of the Ryu SDN controller to provide network identity and flow (traffic classification) metadata. It expects a file called "config.yaml" to be in the same directory containing properly formed YAML

class config.**Config**(*config_dir='config'*, *config_filename='config.yaml'*)
Bases: object

This class is instantiated by nmeta.py and provides methods to ingest the configuration file and provides access to the keys/values that it contains. Config file is in YAML in config subdirectory and is called 'config.yaml'

get_value(*config_key*)
Passed a key and see if it exists in the config YAML. If it does then return the value, if not return 0

10.9 flows module

The flows module is part of the nmeta suite

It provides an abstraction for conversations (flows), using a MongoDB database for storage and data retention maintenance.

Flows are identified via an indexed bi-directionally-unique hash value, derived from IP-value-ordered 5-tuple (source and destination IP addresses, IP protocol and transport source and destination port numbers).

Ingesting a packet puts the flows object into the context of the packet that flow belongs to, and updates the database object for that flow with information from the current packet.

There are various methods (see class docstring) that provide views into the state of the flow.

class flows.**Flow**(*config*)
Bases: baseclass.BaseClass

An object that represents a flow that we are classifying

Intended to provide an abstraction of a flow that classifiers can use to make determinations without having to understand implementations such as database lookups etc.

Be aware that this module is not very mature yet. It does not cover some basic corner cases such as packet retransmissions and out of order or missing packets.

Read a packet_in event into flows (assumes class instantiated as an object called 'flow'):

flow.ingest_packet(dpid, in_port, pkt, timestamp)

Variables available for Classifiers (assumes class instantiated as an object called 'flow'):

Variables for the current packet:

flow.packet.flow_hash The hash of the 5-tuple of the current packet

flow.packet.packet_hash The hash of the current packet used for deduplication. It is an indexed uni-directionally packet identifier, derived from ip_src, ip_dst, proto, tp_src, tp_dst, tp_seq_src, tp_seq_dst

flow.packet.dpid The DPID that the current packet was received from via a Packet-In message

flow.packet.in_port The switch port that the current packet was received on before being sent to the controller

flow.packet.timestamp The time in datetime format that the current packet was received at the controller

flow.packet.length Length in bytes of the current packet on wire

flow.packet.eth_src Ethernet source MAC address of current packet

flow.packet.eth_dst Ethernet destination MAC address of current packet

flow.packet.eth_type Ethertype of current packet in decimal

flow.packet.ip_src IP source address of current packet

flow.packet.ip_dst IP destination address of current packet

flow.packet.proto IP protocol number of current packet

flow.packet.tp_src Source transport-layer port number of current packet

flow.packet.tp_dst Destination transport-layer port number of current packet

flow.packet.tp_flags Transport-layer flags of the current packet

flow.packet.tp_seq_src Source transport-layer sequence number (where existing) of current packet

flow.packet.tp_seq_dst Destination transport-layer sequence number (where existing) of current packet

flow.packet.payload Payload data of current packet

flow.packet.tcp_fin() True if TCP FIN flag is set in the current packet

flow.packet.tcp_syn() True if TCP SYN flag is set in the current packet

flow.packet.tcp_rst() True if TCP RST flag is set in the current packet

flow.packet.tcp_psh() True if TCP PSH flag is set in the current packet

flow.packet.tcp_ack() True if TCP ACK flag is set in the current packet

flow.packet.tcp_urg() True if TCP URG flag is set in the current packet

flow.packet.tcp_ece() True if TCP ECE flag is set in the current packet

flow.packet.tcp_cwr() True if TCP CWR flag is set in the current packet

Variables for the whole flow:

flow.packet_count() Unique packets registered for the flow

flow.client() The IP that is the originator of the flow (if known, otherwise 0)

flow.server() The IP that is the destination of the flow (if known, otherwise 0)

flow.packet_direction() c2s (client to server) or s2c directionality based on first observed packet direction in the flow. Source of first packet in flow is assumed to be the client

flow.max_packet_size() Size of largest packet in the flow

flow.max_interpacket_interval() TBD

flow.min_interpacket_interval() TBD

Variables for the whole flow relating to classification:

classification.TBD

Challenges (not handled - yet):

- duplicate packets due to retransmissions or multiple switches in path
- IP fragments
- Flow reuse - TCP source port reused

class `Classification` (*flow_hash*, *clsfn*, *time_limit*)
Bases: `object`

An object that represents an individual traffic classification

`commit`()
Record current state of flow classification into MongoDB classifications collection.

`dbdict`()
Return a dictionary object of traffic classification parameters for storing in the database

class `Flow.Packet`
Bases: `object`

An object that represents the current packet

`dbdict`()
Return a dictionary object of metadata parameters of current packet (excludes payload), for storing in database

`tcp_ack`()
Does the current packet have the TCP ACK flag set?

`tcp_cwr`()
Does the current packet have the TCP CWR flag set?

`tcp_ece`()
Does the current packet have the TCP ECE flag set?

`tcp_fin`()
Does the current packet have the TCP FIN flag set?

`tcp_psh`()
Does the current packet have the TCP PSH flag set?

`tcp_rst`()
Does the current packet have the TCP RST flag set?

`tcp_syn`()
Does the current packet have the TCP SYN flag set?

`tcp_urg`()
Does the current packet have the TCP URG flag set?

Flow.**client**()
The IP that is the originator of the flow (if known, otherwise 0)

Finds first packet seen for the flow_hash within the time limit and returns the source IP

Flow.**ingest_packet**(*dpid*, *in_port*, *pkt*, *timestamp*)
Ingest a packet into the packet_ins collection and put the flow object into the context of the packet. Note that timestamp MUST be in datetime format

Flow.**max_interpacket_interval**()
Return the size of the largest inter-packet time interval in the flow (assessed per direction in flow) as seconds (type float)

Note: c2s = client to server direction s2c = server to client direction

Note: results are slightly inaccurate due to floating point rounding.

Flow.**max_packet_size**()
Return the size of the largest packet in the flow (in either direction)

Flow.**min_interpacket_interval**()
Return the size of the smallest inter-packet time interval in the flow (assessed per direction in flow) as seconds (type float)

Note: c2s = client to server direction s2c = server to client direction

Note: results are slightly inaccurate due to floating point rounding.

Flow.**packet_count**()
Return the number of packets in the flow (counting packets in both directions). This method should deduplicate for where the same packet is received from multiple switches, but is TBD...

Works by retrieving packets from packet_ins database with current packet flow_hash and within flow reuse time limit.

Flow.**packet_direction**()
Return the direction of the current packet in the flow where c2s is client to server and s2c is server to client.

Flow.**server**()
The IP that is the destination of the flow (if known, otherwise 0)

Finds first packet seen for the hash within the time limit and returns the destination IP

Flow.**suppress_flow**()
Set the suppressed attribute in the flow database object to the current packet count so that future suppressions of the same flow can be backed off to prevent overwhelming the controller

10.10 identities module

The identities module is part of the nmeta suite

It provides an abstraction for participants (identities), using a MongoDB database for storage and data retention maintenance.

Identities are identified via TBD....

There are methods (see class docstring) that provide harvesting of identity metadata and various retrieval searches

class identities.**Identities**(*config*)
Bases: baseclass.BaseClass

An object that represents identity metadata

Variables available for Classifiers (assumes class instantiated as an object called 'ident'):

ident.TBD TBD

ident.harvest(pkt, flow.packet) TBD

ident.findbymac(mac_address)

Challenges (not handled - yet):

- TBD

class Identity
Bases: object

An object that represents an individual Identity Indicator

dbdict()
Return a dictionary object of identity metadata parameters for storing in the database

Identities.**findbymac**(*mac_addr*)
TEST FIND BY MAC ADDR DOC TBD

Identities.**findbynode**(*host_name*, *harvest_type='any'*, *regex=False*)
Find by node name Pass it the name of the node to search for. Additionally, can set:

regex=True Treat service_name as a regular expression harvest_type= Specify what type of harvest (i.e. DHCP)

Returns boolean

Identities.**findbyservice**(*service_name*, *harvest_type='any'*, *regex=False*, *ip_address='any'*)
Find by service name Pass it the name of the service to search for. Additionally, can set:

regex=True Treat service_name as a regular expression harvest_type= Specify what type of harvest (i.e. DNS_A) ip_address= Look for specific IP address

Returns boolean

Identities.**harvest**(*pkt*, *flow_pkt*)
Passed a raw packet and packet metadata from flow object. Check a packet_in event and harvest any relevant identity indicators to metadata

Identities.**harvest_arp**(*pkt*, *flow_pkt*)
Harvest ARP identity metadata into database. Passed packet-in metadata from flow object. Check ARP reply and harvest identity indicators to metadata

Identities.**harvest_dhcp**(*flow_pkt*)
Harvest DHCP identity metadata into database. Passed packet-in metadata from flow object. Check LLDP TLV fields and harvest any relevant identity indicators to metadata

Identities.**harvest_dns**(*flow_pkt*)
Harvest DNS identity metadata into database. Passed packet-in metadata from flow object. Check DNS answer(s) and harvest any relevant identity indicators to metadata

Identities.**harvest_lldp**(*flow_pkt*)
Harvest LLDP identity metadata into database. Passed packet-in metadata from flow object. Check LLDP TLV fields and harvest any relevant identity indicators to metadata

identities.**mac_addr**(*address*)
Convert a MAC address to a readable/printable string

10.11 forwarding module

This module is part of the nmeta suite running on top of Ryu SDN controller to provide network identity and flow metadata. It provides methods for forwarding functions.

class forwarding.**Forwarding**(*_config*)
Bases: object

This class is instantiated by nmeta.py and provides methods for making forwarding decisions and transformations to packets.

basic_switch(*event*, *in_port*)
Passed a packet in event and return an output port

10.12 switch_abstraction module

This module is part of the nmeta suite running on top of Ryu SDN controller. It provides functions that abstract the details of OpenFlow calls, including differences between OpenFlow versions where practical

class switch_abstraction.**SwitchAbstract**(*config*)
Bases: baseclass.BaseClass

This class is instantiated by various other modules and provides methods for interacting with switches that are safe to use without need to for the calling program to know calls specific to the version of OpenFlow that the switch runs (where practical...)

add_flow(*datapath*, *match*, *actions*, ***kwargs*)
Add a flow table entry to a switch. Returns 1 for success or 0 for any type of error

Required kwargs are: priority (0) buffer_id (None) idle_timeout (5) hard_timeout (0)

add_flow_eth(*datapath*, *msg*, ***kwargs*)
Add an ethernet (non-IP) flow table entry to a switch. Returns 1 for success or 0 for any type of error Uses Ethertype in match to prevent matching against IPv4 or IPv6 flows

add_flow_ip(*datapath*, *msg*, ***kwargs*)
Add an IP (v4 or v6) flow table entry to a switch. Returns 1 for success or 0 for any type of error Uses IP protocol number to prevent matching on TCP flows

add_flow_tcp(*datapath*, *msg*, ***kwargs*)
Add a TCP flow table entry to a switch. Returns 1 for success or 0 for any type of error

get_actions(*datapath*, *ofv*, *out_port*, *out_queue*)
Passed a datapath, an OpenFlow version an out port, an out queue and flood port # and build and return an appropriate set of actions for this

get_flow_match(*datapath*, *ofproto*, ***kwargs*)
Passed a OF protocol version and a Flow Match keyword arguments dict and return an OF match tailored for the OF version otherwise 0 (false) if compatibility not possible. TBD: validating values...

get_friendly_of_version(*ofproto*)
Passed an OF Protocol object and return a human-friendly version of the protocol revision number

get_in_port(*msg*, *datapath*, *ofproto*)
Passed a msg, datapath and OF protocol version and return the port that the packet came in on (version specific)

packet_out (*datapath*, *msg*, *in_port*, *out_port*, *out_queue*, *nq=0*)
Sends a supplied packet out switch port(s) in specific queue. Set nq=1 if want no queueing specified (i.e. for a flooded packet)

request_switch_desc (*datapath*)
Request that a switch send us it's description data

set_switch_config (*datapath*, *config_flags*, *miss_send_len*)
Set config on a switch including config flags that instruct fragment handling behaviour and miss_send_len which controls the number of bytes sent to the controller when the output port is specified as the controller

set_switch_table_miss (*datapath*, *miss_send_len*, *hw_desc*, *sw_desc*)
Set a table miss rule on table 0 to send packets to the controller. This is required for OF versions higher than v1.0. Do not set on older OpenvSwitch as it causes packets to be sent to controller with no buffer and OpenvSwitch doesn't need this rule as it punts to the controller regardless (contrary to specification?) Note: OVS 2.5.0 doesn't do this, but not sure what point fix was applied to OVS, somewhere between 2.0.2 and 2.5.0

CHAPTER 11

Indices and tables

- genindex
- modindex
- search

a

c

f

i

n

s

t

Symbols

A

B

C

D

E

F

G

H

U

V