nmeta2dpae Documentation

Release 0.3.5

Matthew John Hayes

September 30, 2016

Contents

The nmeta2 project is a research platform for distributed scalable traffic classification on Software Defined Networking (SDN). Read More

Contents:

CHAPTER 1

Introduction

The nmeta2dpae project is a sub-system of the nmeta2 project

1.1 Distributed System

Nmeta2dpae is the code that runs the distributed heavy-lifting work of traffic classification on auxiliary devices, called a Data Plane Auxiliary Engines (DPAE), that scale horizontally.

1.2 Limitations

Nmeta2dpae code is under construction, so a number of features are not implemented yet, or not finished.

1.3 Feature Enhancement Wishlist

See Issues for list of enhancements and bugs

1.4 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.

1.5 Disclaimer

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

1.6 How to Contribute to the Code

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

Install

This guide is for installing nmeta2dpae on Ubuntu OS.

2.1 Pre-Work

2.1.1 Ensure packages are up-to-date

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

2.1.2 Install Python pip

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

2.1.3 Install git

Install git and git-flow for software version control:

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

2.2 Install Packages Required by nmeta

2.2.1 Install coloredlogs

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

```
sudo pip install coloredlogs
```

2.2.2 Install dpkt Python Packet Library

Install dpkt for parsing packets:

```
sudo pip install dpkt
```

2.2.3 Install scapy

```
sudo pip install scapy
```

2.2.4 Install pytest

Pytest is used to run unit tests:

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

2.2.5 Install YAML

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

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

2.2.6 Install simplejson

```
sudo pip install simplejson
```

2.2.7 Install mock

```
sudo pip install -U mock
```

2.3 Install MongoDB

MongoDB is the database used by nmeta2. Install MongoDB as per their instructions :

Import the MongoDB public GPG Key:

```
sudo apt-key adv --keyserver hkp://keyserver.ubuntu.com:80 --recv EA312927
```

Create a list file for MongoDB:

```
echo "deb http://repo.mongodb.org/apt/ubuntu trusty/mongodb-org/3.2 multiverse" | sudo tee /etc/apt/s
```

Reload local package database:

```
sudo apt-get update
```

Install MongoDB:

```
sudo apt-get install -y mongodb-org
```

Add pymongo for a Python API into MongoDB:

```
sudo apt-get install build-essential python-dev
sudo pip install pymongo
```

Turn on smallfiles to cope with small file system size:

```
sudo vi /etc/mongod.conf
```

Add this to the storage section of the config:

```
mmapv1:
  smallFiles: true
```

Start MongoDB (if required) with:

```
sudo service mongod start
```

2.4 Install nmeta2dpae

Clone nmeta2dpae

```
cd
git clone https://github.com/mattjhayes/nmeta2dpae.git
```

2.5 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 nmeta2dpae:
alias nm2="sudo python ~/nmeta2dpae/nmeta2dpae/nmeta2dpae.py"
#
# Run tests on nmeta2dpae:
alias nm2t="cd ~/nmeta2dpae/test/; py.test"
```

2.5.1 Re-read the Aliases

Read the aliases file in so that new command is available for use:

```
. ~/.bashrc
```

2.6 Edit Config

Edit the config file ~/nmeta2dpae/nmeta2dpae/config/config.yaml and update values as appropriate. You should check:

- URL for nmeta2 under key nmeta_controller_address
- Which interfaces should sniff under key sniff_if_names

- MongoDB settings under keys mongo_addr and mongo_port

2.7 Create Custom Classifiers

Custom classifiers can be installed into the ~/nmeta2dpae/nmeta2dpae/classifiers directory. They operate per packet and are passed a flow class object that has variables and methods that are in the context of the current packet and the flow that it belongs to. Check out flow.py for more information. Custom classifiers are called by declaring them in main_policy.yaml in nmeta2 on the controller.

2.8 Run nmeta2dpae

```
nm2
```

CHAPTER 3

Module Documentation

3.1 nmeta2dpae package

3.1.1 Submodules

3.1.2 nmeta2dpae.config module

This module is part of the nmeta2 suite . It represents the configuration data for a Data Plane Auxiliary Engine (DPAE). . It expects a file called "config.yaml" to be in a subdirectory called config, and this file to contain properly formed YAML

class nmeta2dpae.config.**Config**
Bases: object

This class is instantiated by nmeta_dpae.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

3.1.3 nmeta2dpae.controlchannel module

This module is part of the nmeta2 suite . It is provides control channel services between the nmeta Data Plane Auxiliary Engine (DPAE) and the OpenFlow controller using REST API calls

class nmeta2dpae.controlchannel.**ControlChannel**(*_nmeta2dpae*, *_config*, *if_name*, *dp*)
Bases: object

This class is instantiated by nmeta_dpae.py and provides methods to interact with the nmeta control plane

get_policy(*location*)
Get the a policy from the Controller (YAML in string format)

keepalive(*event_flag*, *location*, *if_name*)
Do regular keepalive polls to the DPAE to check if is still available, in dedicated process. If keepalive fails, then set an event flag for parent process.

phase1(*api_base*, *if_name*)
Phase 1 (global to DPAE) connection to the control plane, as an active data plane auxiliary device

phase2(*api_base*, *if_name*, *dpae2ctrl_mac*, *ctrl2dpae_mac*, *dpae_ethertype*)
Phase 2 (per DPAE sniffing interface) switch/port discovery

phase3 (*api_base*, *if_name*, *dpae2ctrl_mac*, *ctrl2dpae_mac*, *dpae_ethertype*)
Phase 3 (per DPAE sniffing interface) confirmation of sniffing packets

phase4 (*api_base*, *if_name*)
Phase 4 (per DPAE sniffing interface) Negotiate what services will be run by the DPAE

tc_advise_controller (*location*, *tc_result*)
Pass Traffic Classification (TC) information to the controller via the API

tc_start (*location*)
Tell the Controller to start sending us packets that need traffic classification

class nmeta2dpae.controlchannel.**JSON_Body** (*req_body*)
Bases: object

Represents a JSON-encoded body of an HTTP request. Doesn't do logging, but does set .error when things don't go to plan with a friendly message.

decode (*req_body*)
Passed an allegedly JSON body and see if it decodes. Set error variable for exceptions

validate (*key_list*)
Passed a list of keys and check that they exist in the JSON. If they don't return 0 and set error to description of first missing key that was found

3.1.4 nmeta2dpae.dp module

This module is part of the nmeta2 suite . It provides an object for data plane coordination services . Version 2.x Toulouse Code

class nmeta2dpae.dp.**DP** (*_config*)
Bases: object

This class is instantiated by nmeta2_dpae.py and provides methods to run the data plane services.

dp_discover (*queue*, *if_name*, *dpae2ctrl_mac*, *ctrl2dpae_mac*, *dpae_ethertype*, *timeout*, *uuid_dpae*, *uuid_controller*)
Data plane service for DPAE Join Discover Packet Sniffing

dp_run (*interplane_queue*, *tc_policy*, *if_name*)
Run Data Plane (DP) Traffic Classification for an interface

class nmeta2dpae.dp.**JSON_Body** (*req_body*)
Bases: object

Represents a JSON-encoded body of an HTTP request. Doesn't do logging, but does set .error when things don't go to plan with a friendly message.

decode (*req_body*)
Passed an allegedly JSON body and see if it decodes. Set error variable for exceptions

validate (*key_list*)
Passed a list of keys and check that they exist in the JSON. If they don't return 0 and set error to description of first missing key that was found

3.1.5 nmeta2dpae.flow module

This module is part of the nmeta2 suite . It provides an abstraction for a TCP flow that links to a MongoDB database and changes to the context of the flow that a supplied packet belongs to . Version 2.x Toulouse Code

class nmeta2dpae.flow.**Flow**(*logger*, *mongo_addr*, *mongo_port*)
Bases: object

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.

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

Variables for the current packet:

flow.ip_src IP source address of latest packet in flow

flow.ip_dst IP dest address of latest packet in flow

flow.tcp_src TCP source port of latest packet in flow

flow.tcp_dst TCP dest port of latest packet in flow

flow.tcp_seq TCP sequence number of latest packet in flow

flow.tcp_acq TCP acknowledgement number of latest packet in flow

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

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

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

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

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

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

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

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

flow.payload Payload of TCP of latest packet in flow

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

flow.packet_direction c2s (client to server) or s2c directionality based on first observed packet having SYN or SYN+ACK flag, otherwise client assumed as source IP of first packet and verified_direction set to 0 (i.e. don't trust packet_direction unless verified_direction is set)

Variables for the whole flow:

flow.verified_direction Describes how the directionality of the flow was ascertained. Values can be verified-SYN, verified-SYNACK or 0 (unverified)

flow.finalised A classification has been made

flow.suppressed The flow packet count number when a request was made to controller to not see further packets in this flow. 0 is not suppressed

flow.packet_count Unique packets registered for the flow

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

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

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

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

flow.max_interpacket_interval() TBD

flow.min_interpacket_interval() TBD

Challenges:

- duplicate packets
- IP fragments (not handled)
- Flow reuse - TCP source port reused (not handled - yet)

ingest_packet(*pkt*, *pkt_receive_timestamp*)
Ingest a packet and put the flow object into the context of the flow that the packet belongs to.

max_interpacket_interval()
Return the size of the largest inter-packet time interval in the flow (assessed per direction in flow). . Note: slightly inaccurate due to floating point rounding.

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

min_interpacket_interval()
Return the size of the smallest inter-packet time interval in the flow (assessed per direction in flow) . Note: slightly inaccurate due to floating point rounding.

set_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

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?

3.1.6 nmeta2dpae.nmeta2dpae module

This module is part of the nmeta2 suite . nmeta Data Plane Auxiliary Engine (DPAE) Used as an auxilary data plane component for functions such as offloading packet-intensive traffic classification from the controller.

class nmeta2dpae.nmeta2dpae.**DPAE**
Bases: object

This class provides methods for a Data Plane Auxiliary Engine (DPAE), an auxiliary entity that provides services to nmeta.

cp_run(*if_name*, *controlchannel*, *location*)
Run Control Plane (CP) Traffic Classification for an interface

per_interface(*if_name*)
Run per interface that sniffing will run on as separate process

run()
Run the DPAE instance

3.1.7 nmeta2dpae.sniff module

This module is part of the nmeta2 suite . It provides packet sniffing services

class nmeta2dpae.sniff.**Ifreq**
Bases: _ctypes.Structure

Class used in setting Ethernet interface promiscuous mode

ifr_flags
Structure/Union member

ifr_ifrn
Structure/Union member

class nmeta2dpae.sniff.**Sniff**(*_config*, *tc*)
Bases: object

This class is instantiated by nmeta_dpae.py and provides methods to sniff and process inbound packets on a given interface

discover_confirm(*if_name*, *dpae2ctrl_mac*, *ctrl2dpae_mac*, *dpae_ethertype*, *timeout*)
This function processes sniffs for a discover confirm packet and returns 1 if seen and valid, otherwise 0 after expiry of timeout period

get_promiscuous_mode(*if_name*)
Return the promiscuous mode of an interface. 1 is promiscuous mode enabled 0 is promiscuous mode disabled

set_promiscuous_mode(*if_name*)
Set a given Ethernet interface to promiscuous mode so that it can receive packets destined for any MAC address.

sniff_run(*if_name*, *tc*, *tc_policy*, *queue*)
This function sniffs packets from a NIC. It passes the packets to the tc module for classification and returns any TC results to parent process via a queue.

In active mode it also sends the processed packet back to the switch

nmeta2dpae.sniff.**mac_addr**(*address*)
Convert a MAC address to a readable/printable string

3.1.8 nmeta2dpae.tc module

This module is part of the nmeta2 suite . It provides an object for traffic classification and includes ingesting the policy from YAML and checking packets against policy, calling appropriate classifiers and returning actions. . Version 2.x Toulouse Code

class nmeta2dpae.tc.**TC**(*_config*)
Bases: object

This class is instantiated by nmeta2_dpae.py and provides methods to ingest the policy as yaml and check packets against policy, calling appropriate classifiers and returning actions.

classify_dpkt(*pkt*, *pkt_receive_timestamp*, *if_name*)
Perform traffic classification on a packet using dpkt for packet parsing

classify_dpkt_wrapper(*pkt*, *pkt_receive_timestamp*, *if_name*)
Used to catch and handle exceptions in classify_dpkt otherwise it can just hang with no explaination... TBD: turn this into a decorator...

instantiate_classifiers(*_classifiers*)
Dynamically import and instantiate classes for any dynamic classifiers specified in the controller nmeta2 main_policy.yaml . Passed a list of tuples of classifier type / classifer name . Classifier modules live in the 'classifiers' subdirectory .

nmeta2dpae.tc.**mac_addr**(*address*)
Convert a MAC address to a readable/printable string

3.1.9 nmeta2dpae.tc_policy_dpae module

This module is part of nmeta Data Plane Auxiliary Engine (DPAE) . It is used to contain the Traffic Classification (TC) policy and provide methods and direct variables to access it . Version 2.x Toulouse Code

class nmeta2dpae.tc_policy_dpae.**TCPolicy**(*_config*)
Bases: object

This class is instantiated by nmeta2.py and provides methods to ingest the policy file main_policy.yaml and validate that it is correctly structured

get_id_flag(*if_name*, *id_key*)
Get a value for an Identity Indicator harvesting flag

get_tc_classifiers(*if_name*)
Return a list of traffic classifiers that should be run against ingress packets on a sniff interface. Each entry is a tuple of type (statistical or payload) and classifier name, example: [('statistical', 'statistical_qos_bandwidth_1')]

ingest_main_policy(*main_policy_text*, *if_name*)
Turn a plain text main policy file object into a YAML object and store it as a class variable

ingest_optimised_rules(*opt_rules_text*, *if_name*)
Turn a plain optimised TC rules file object into a YAML object and store it as a class variable

tc_mode(*if_name*)
Return the tc mode for the policy (active or passive)

3.1.10 Module contents

3.2 nmeta2dpae.classifiers package

3.2.1 Submodules

3.2.2 nmeta2dpae.classifiers.payload_uri_1 module

This module is part of the nmeta2 suite . It defines a custom traffic classifier . To create your own custom classifier, copy this example to a new file in the same directory and update the code as required. Call it from nmeta by specifying the name of the file (without the .py) in main_policy.yaml . Classifiers are called per packet, so performance is important .

class nmeta2dpae.classifiers.payload_uri_1.**Classifier**(*logger*)
Bases: object

A custom classifier module for import by nmeta2

classifier(*flow*)
A really basic HTTP URI classifier to demonstrate ability to differentiate based on a payload characteristic. . This method is passed a Flow class object that holds the current context of the flow . It returns a dictionary specifying a key/value of QoS treatment to take (or not if no classification determination made). . Only works on TCP.

3.2.3 nmeta2dpae.classifiers.statistical_qos_bandwidth_1 module

This module is part of the nmeta2 suite . It defines a custom traffic classifier . To create your own custom classifier, copy this example to a new file in the same directory and update the code as required. Call it from nmeta by specifying the name of the file (without the .py) in main_policy.yaml . Classifiers are called per packet, so performance is important .

class nmeta2dpae.classifiers.statistical_qos_bandwidth_1.**Classifier**(*logger*)
Bases: object

A custom classifier module for import by nmeta2

classifier(*flow*)
A really basic statistical classifier to demonstrate ability to differentiate 'bandwidth hog' flows from ones that are more interactive so that appropriate classification metadata can be passed to QoS for differential treatment. . This method is passed a Flow class object that holds the current context of the flow . It returns a dictionary specifying a key/value of QoS treatment to take (or not if no classification determination made). . Only works on TCP.

3.2.4 Module contents

CHAPTER 4

Indices and tables

- genindex
- modindex
- search

n

P

R

S

T

V