Tensor Documentation

Release 0.0.6

Colin Alston

February 07, 2015

Contents

Tensor is a modular gateway and event router for Riemann, built using the Twisted framework.

Contents:

CHAPTER 1

Getting started

1.1 Installation

Tensor can be installed from PyPi with pip

```
$ pip install tensor
```

This will also install Twisted, protobuf and PyYAML

Or you can use the .deb package

```
$ aptitude install python-twisted python-protobuf python-yaml
$ wget https://github.com/calston/tensor/releases/download/0.0.7/tensor_0.0.7_amd64.deb
$ dpkg -i tensor_0.0.7_amd64.deb
```

This also gives you an init script and default config in /etc/tensor/

1.2 Creating a configuration file

Tensor uses a simple YAML configuration file

The first basic requirements are the Riemann server and port (defaults to localhost:5555) and the queue interval:

```
server: localhost
port: 5555
interval: 1.0
proto: udp
```

Tensors checks are Python classes (called sources) which are instantiated with the configuration block which defines them. Rather than being one-shot scripts, a source object remains in memory with its own timer which adds events to a queue. The *interval* defined above is the rate at which these events are rolled up into a message and sent to Riemann.

It is important then that *interval* is set to a value appropriate to how frequently you want to see them in Riemann, as well as the rate at which they collect metrics from the system. All *interval* attributes are floating point in seconds, this means you can check (and send to Riemann) at rates well below 1 second.

You can configure multiple outputs which receive a copy of every message for example

```
outputs:
    - output: tensor.outputs.riemann.RiemannUDP
      server: localhost
      port: 5555
```

If you enable multiple outputs then the *server*, *port* and *proto* options will go un-used and the default Riemann TCP transport won't start.

You can configure as many outputs as you like, or create your own.

To configure the basic CPU usage source add it to the *sources* list in the config file

```
sources:
    - service: cpu
      source: tensor.sources.linux.basic.CPU
      interval: 1.0
      warning: {
        cpu: "> 0.5"
      }
      critical: {
        cpu: "> 0.8"
      }
```

This will measure the CPU from /proc/stat every second, with a warning state if the value exceeds 50%, and a critical state if it exceeds 80%

The *service* attribute can be any string you like, populating the *service* field in Riemann. The logical expression to raise the state of the event is (eg. critical) is assigned to a key which matches the service name.

Sources may return more than one metric, in which case it will add a prefix to the service. The state expression must correspond to that as well.

For example, the Ping check returns both latency and packet loss:

```
service: googledns
source: tensor.sources.network.Ping
interval: 60.0
destination: 8.8.8.8
critical: {
    googledns.latency: "> 100",
    googledns.loss: "> 0"
}
```

This will ping *8.8.8.8* every 60 seconds and raise a critical alert for the latency metric if it exceeds 100ms, and the packet loss metric if there is any at all.

1.3 Starting Tensor

To start Tensor, simply use twistd to run the service and pass a config file:

```
twistd -n tensor -c tensor.yml
```

CHAPTER 2

Sources

2.1 Introduction

Sources are Python objects which subclass `tensor.objects.Source`. They are constructed with a dictionary parsed from the YAML configuration block which defines them, and as such can read any attributes from that either optional or mandatory.

Since sources are constructed at startup time they can retain any required state, for example the last metric value to report rates of change or for any other purpose. However since a Tensor process might be running many checks a source should not use an excessive amount of memory.

The *source* configuration option is passed a string representing an object in much the same way as you would import it in a python module. The final class name is split from this string. For example specifying:

```
source: tensor.sources.network.Ping
```

is equivalent to:

```
from tensor.sources.network import Ping
```

2.2 Writing your own sources

A source class must subclass `tensor.objects.Source` and also implement the interface `tensor.interfaces.ITensorSource`

The source must have a *get* method which returns a `tensor.objects.Event` object. The Source parent class provides a helper method *createEvent* which performs the metric level checking (evaluating the simple logical statement in the configuration), sets the correct service name and handles prefixing service names.

A "Hello world" source:

```
from zope.interface import implements

from tensor.interfaces import ITensorSource
from tensor.objects import Source

class HelloWorld(Source):
    implements(ITensorSource)

    def get(self):
        return self.createEvent('ok', 'Hello world!', 0)
```

To hold some state, you can re-implement the *__init__* method, as long as the arguments remain the same.

Extending the above example to create a simple flip-flop metric event:

```
from zope.interface import implements

from tensor.interfaces import ITensorSource
from tensor.objects import Source

class HelloWorld(Source):
    implements(ITensorSource)

    def __init__(self, *a):
        Source.__init__(self, *a)
        self.bit = False

    def get(self):
        self.bit = not self.bit
        return self.createEvent('ok', 'Hello world!', self.bit and 0.0 or 1.0)
```

You could then place this in a Python module like *hello.py* and as long as it's in the Python path for Tensor it can be used as a source with *hello.HelloWorld*

2.3 Handling asynchronous tasks

Since Tensor is written using the Twisted asynchronous framework, sources can (and in most cases *must*) make full use of it to implement network checks, or execute other processes.

The simplest example of a source which executes an external process is the ProcessCount check:

```
from zope.interface import implements

from twisted.internet import defer

from tensor.interfaces import ITensorSource
from tensor.objects import Source
from tensor.utils import fork

class ProcessCount(Source):
    implements(ITensorSource)

    @defer.inlineCallbacks
    def get(self):
        out, err, code = yield fork('/bin/ps', args=('-e',))

        count = len(out.strip('\n').split('\n'))

        defer.returnValue(
            self.createEvent('ok', 'Process count %s' % (count), count)
        )
```

For more information please read the Twisted documentation at https://twistedmatrix.com/trac/wiki/Documentation

The `tensor.utils.fork()` method returns a deferred which can timeout after a specified time.

2.4 Thinking outside the box

Historically monitoring systems are poorly architected, and terribly inflexible. To demonstrate how Tensor offers a different concept to the boring status quo it's interesting to note that there is nothing preventing you from starting a listening service directly within a source which processes and relays events to Riemann implementing some protocol.

Here is an example of a source which listens for TCP connections to port 8000, accepting any number on a line and passing that to the event queue:

```python
from twisted.internet.protocol import Factory
from twisted.protocols.basic import LineReceiver
from twisted.internet import reactor

from zope.interface import implements

from tensor.interfaces import ITensorSource
from tensor.objects import Source

class Numbers(LineReceiver):
    def __init__(self, source):
        self.source = source

    def lineReceived(self, line):
        """
        Send any numbers received back to the Tensor queue
        """
        print repr(line)
        try:
            num = float(line)
            self.source.queueBack(
                self.source.createEvent('ok', 'Number: %s' % num, num)
            )
        except:
            pass

class NumbersFactory(Factory):
    def __init__(self, source):
        self.source = source

    def buildProtocol(self, addr):
        return Numbers(self.source)

class NumberProxy(Source):
    implements(ITensorSource)

    def startTimer(self):
        # Override starting the source timer, we don't need it
        f = NumbersFactory(self)
        reactor.listenTCP(8000, f)

    def get(self):
        # Implement the get method, but we can ignore it
        pass
```

CHAPTER 3

Outputs

3.1 Introduction

Outputs are Python objects which subclass `tensor.objects.Output`. They are constructed with a dictionary parsed from the YAML configuration block which defines them, and as such can read any attributes from that either optional or mandatory.

Since outputs are constructed at startup time they can retain any required state. A copy of the queue is passed to all **:method:‘tensor.objects.Output.eventsReceived‘** calls which happen at each queue *interval* config setting as the queue is emptied. This list of `tensor.objects.Event` objects must not be altered by the output.

The *output* configuration option is passed a string representing an object the same way as *sources* configurations are

```
output: tensor.sources.network.Ping
```

3.2 Writing your own outputs

An output clas should subclass `tensor.objects.Output`.

The output can implement a *createClient* method which starts the output in whatever way necessary and can be a deferred. The output must also have a *eventsReceived* method which takes a list of `tensor.objects.Event` objects and process them accordingly, it can also be a deferred.

An example logging source:

```
from twisted.internet import reactor, defer
from twisted.python import log

from tensor.objects import Output

class Logger(Output):
    def eventsReceived(self, events):
        log.msg("Events dequeued: %s" % len(events))
```

If you save this as *test.py* the basic configuration you need is simply

```
outputs:
    - output: tensor.outputs.riemann.RiemannUDP
      server: localhost
      port: 5555

    - output: test.Logger
```

You should now see how many events are exiting in the Tensor log file

```
2014-10-24 15:35:27+0200 [-] Starting protocol <tensor.protocol.riemann.RiemannUDP object at 0x7f3b5a
2014-10-24 15:35:28+0200 [-] Events dequeued: 7
2014-10-24 15:35:29+0200 [-] Events dequeued: 2
2014-10-24 15:35:30+0200 [-] Events dequeued: 3
```

CHAPTER 4

Example configurations

4.1 Replacing Munin

The first step is to create a TRIG stack (Tensor Riemann InfluxDB Grafana).

4.1.1 Step 1: Install Riemann

```
$ wget http://aphyr.com/riemann/riemann_0.2.6_all.deb
$ aptitude install openjdk-7-jre
$ dpkg -i riemann_0.2.6_all.deb
```

4.1.2 Step 2: Install InfluxDB

```
$ wget http://s3.amazonaws.com/influxdb/influxdb_latest_amd64.deb
$ sudo dpkg -i influxdb_latest_amd64.deb
```

Start InfluxDB, then quickly change the root/root default password because it also defaults to listening on all interfaces and apparently this is not important enough for them to fix.

Create a *riemann* and *grafana* database, and some users for them

```
$ curl -X POST 'http://localhost:8086/db?u=root&p=root' \
  -d '{"name": "riemann"}'
$ curl -X POST 'http://localhost:8086/db?u=root&p=root' \
  -d '{"name": "grafana"}'
$ curl -X POST 'http://localhost:8086/db/riemann/users?u=root&p=root' \
  -d '{"name": "riemann", "password": "riemann"}'
$ curl -X POST 'http://localhost:8086/db/grafana/users?u=root&p=root' \
  -d '{"name": "grafana", "password": "grafana"}'
```

4.1.3 Step 3: Install Grafana

```
$ aptitude install nginx
$ mkdir /var/www
$ cd /var/www
$ wget http://grafanarel.s3.amazonaws.com/grafana-1.8.1.tar.gz
$ tar -zxf grafana-1.8.1.tar.gz
$ mv grafana-1.8.1 grafana
```

Now we must create an nginx configuration in */etc/nginx/sites-enabled*.

You can use something like this

```
server {
    listen 80;
    server_name <your hostname>;
    access_log /var/log/nginx/grafana-access.log;
    error_log /var/log/nginx/grafana-error.log;

    location / {
        alias /var/www/grafana/;
        index index.html;
        try_files $uri $uri/ /index.html;
    }
}
```

Next we need a configuration file for grafana. Open */var/www/grafana/config.js* and use the following configuration

```
define(['settings'],
function (Settings) {
  return new Settings({
    datasources: {
      influxdb: {
        type: 'influxdb',
        url: "http://<your hostname>:8086/db/riemann",
        username: 'riemann',
        password: 'riemann',
      },
      grafana: {
        type: 'influxdb',
        url: "http://<your hostname>:8086/db/grafana",
        username: 'grafana',
        password: 'grafana',
        grafanaDB: true
      },
    },
    search: {
      max_results: 20
    },
    default_route: '/dashboard/file/default.json',
    unsaved_changes_warning: true,
    playlist_timespan: "1m",
    admin: {
      password: ''
    },
    window_title_prefix: 'Grafana - ',
    plugins: {
      panels: [],
      dependencies: [],
    }
  });
});
```

4.1.4 Step 4: Glue things together

Lets start by configuring Riemann to talk to InfluxDB. This is the full /etc/riemann/riemann.config file.

```
; -*- mode: clojure; -*-
; vim: filetype=clojure
(require 'capacitor.core)
(require 'capacitor.async)
(require 'clojure.core.async)

(defn make-async-influxdb-client [opts]
    (let [client (capacitor.core/make-client opts)
          events-in (capacitor.async/make-chan)
          resp-out (capacitor.async/make-chan)]
        (capacitor.async/run! events-in resp-out client 100 10000)
        (fn [series payload]
            (let [p (merge payload {
                    :series series
                    :time   (* 1000 (:time payload)) ;; s → ms
                })]
                (clojure.core.async/put! events-in p)))))

(def influx (make-async-influxdb-client {
        :host     "localhost"
        :port     8086
        :username "riemann"
        :password "riemann"
        :db       "riemann"
    }))

(logging/init {:file "/var/log/riemann/riemann.log"})

; Listen on the local interface over TCP (5555), UDP (5555), and websockets
; (5556)
(let [host "0.0.0.0"]
  (tcp-server {:host host})
  (udp-server {:host host})
  (ws-server  {:host host}))

(periodically-expire 60)

(let [index (index)]
  (streams
        index

        (fn [event]
            (let [series (format "%s.%s" (:host event) (:service event))]
                (influx series {
                    :time  (:time event)
                    :value (:metric event)
                })))))
```

You're pretty much done at this point, and should see the metrics from the Riemann server process if you open up Grafana and look through the query builder.

4.1.5 Step 5: Using Tensor to retrieve stats from munin-node

First of all, install Tensor

```
$ pip install tensor
```

Next create /etc/tensor and a *tensor.yml* file in that directory.

The *tensor.yml* config file should look like this

```
ttl: 60.0
interval: 1.0

outputs:
    - output: tensor.outputs.riemann.RiemannTCP
      port: 5555
      server: <riemann server>

# Sources
sources:
    - service: mymunin
      source: tensor.sources.munin.MuninNode
      interval: 60.0
      ttl: 120.0
      critical: {
        mymunin.system.load.load: "> 2"
      }
```

This configures Tensor to connect to the munin-node on the local machine and retrieve all configured plugin values. You can create critical alert levels by setting the dot separated prefix for the service name and munin plugin.

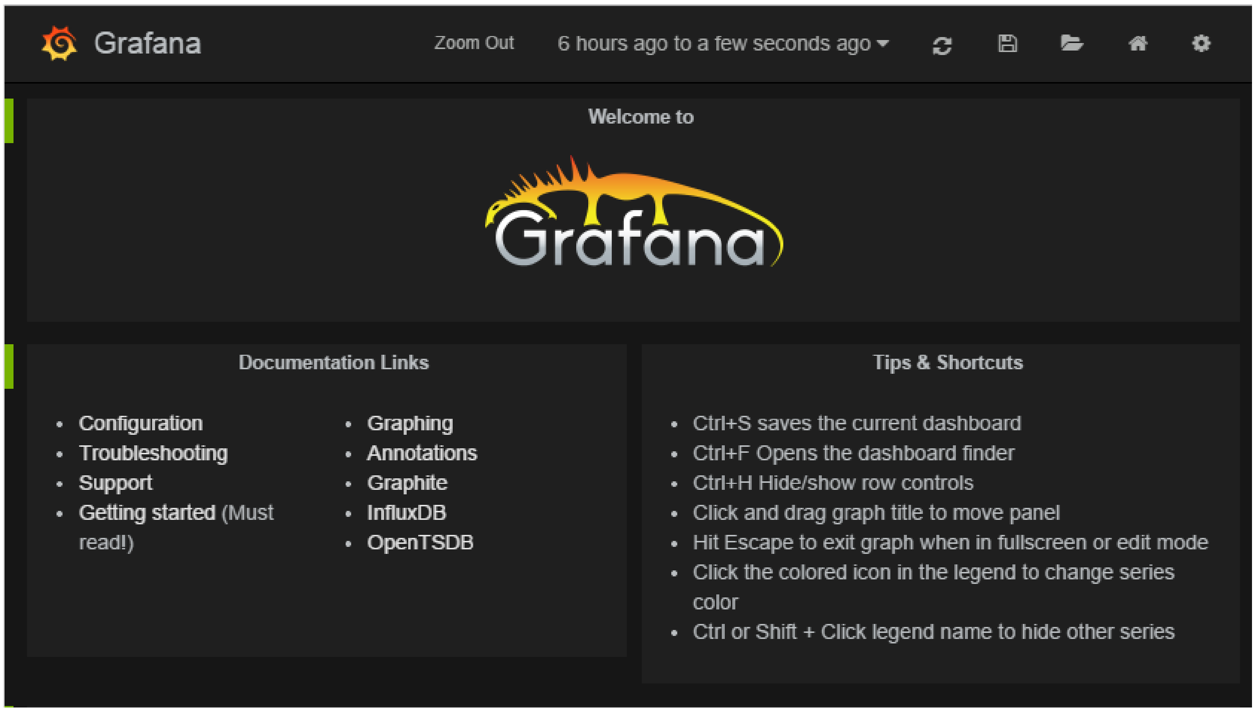
You can now start Tensor

```
$ twistd -n tensor -c /etc/tensor/tensor.yml
2014-10-22 13:30:38+0200 [-] Log opened.
2014-10-22 13:30:38+0200 [-] twistd 14.0.2 (/home/colin/riemann-tensor/ve/bin/python 2.7.6) starting
2014-10-22 13:30:38+0200 [-] reactor class: twisted.internet.epollreactor.EPollReactor.
2014-10-22 13:30:38+0200 [-] Starting factory <tensor.protocol.riemann.RiemannClientFactory instance
```

This pretty much indiciates everything is alright, or else we'd see quickly see some errors.

Next we will add some graphs to Grafana

4.1.6 Step 6: Creating graphs in Grafana



Documentation Links

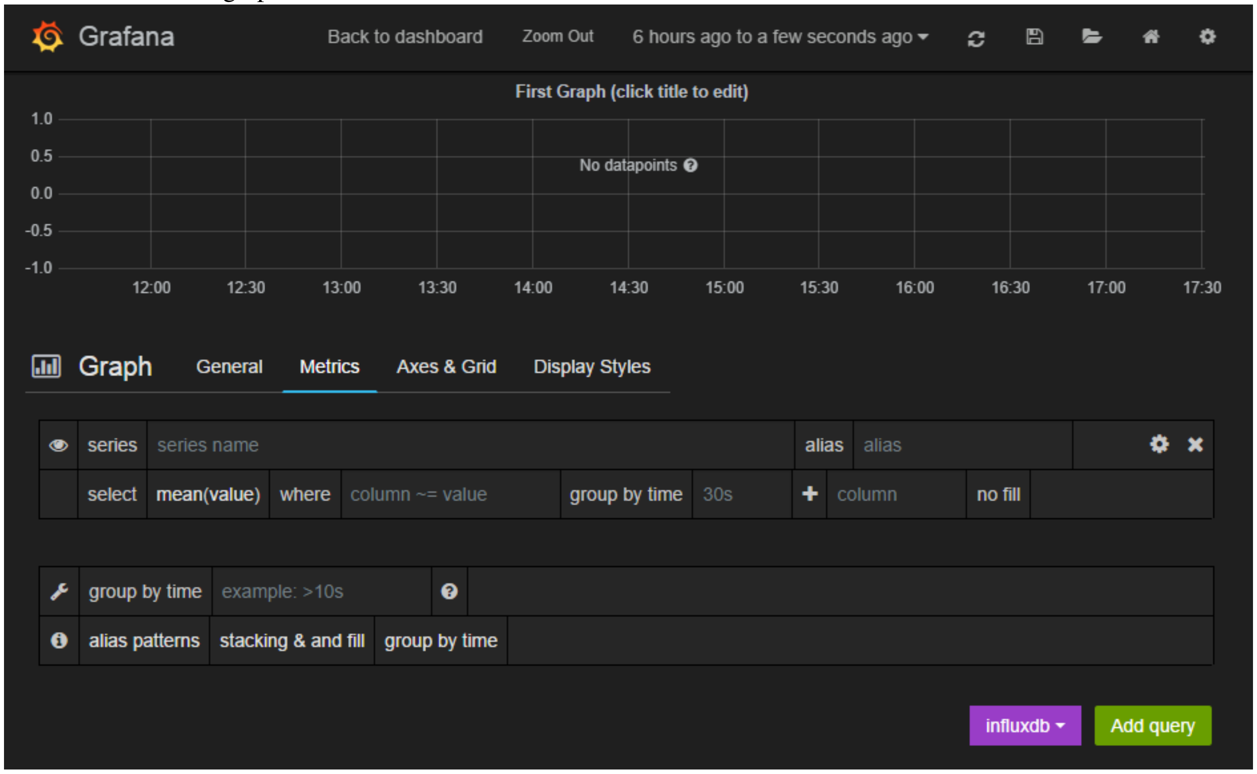
- Configuration
- Troubleshooting
- Support
- Getting started (Must read!)
- Graphing
- Annotations
- Graphite
- InfluxDB
- OpenTSDB

Tips & Shortcuts

- Ctrl+S saves the current dashboard
- Ctrl+F Opens the dashboard finder
- Ctrl+H Hide/show row controls
- Click and drag graph title to move panel
- Hit Escape to exit graph when in fullscreen or edit mode
- Click the colored icon in the legend to change series color
- Ctrl or Shift + Click legend name to hide other series

Click on the green row tag on the left, and delete all but the last row. This will leave you with an empty graph.

Click the title of the graph, then click *Edit*.



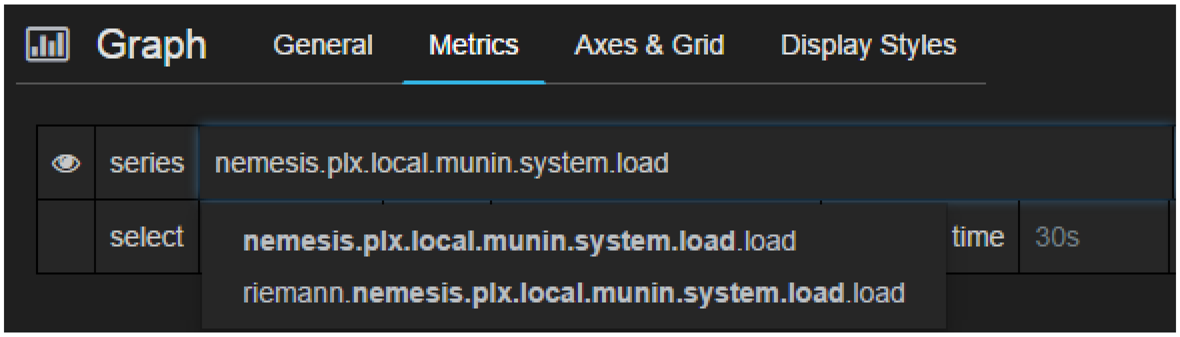
series | series name | alias | alias

select | mean(value) | where | column ~= value | group by time | 30s | + | column | no fill

group by time | example: >10s

alias patterns | stacking & and fill | group by time

In the edit screen the Metrics tab will be open already. Now we can add our munin metrics. If you start typing in the *series* field you should see your hosts and metrics autocomplete.



```
series  nemesis.plx.local.munin.system.load
select  nemesis.plx.local.munin.system.load.load
        riemann.nemesis.plx.local.munin.system.load.load
time    30s
```

Many Munin metrics are *counter* types which are usually converted to a rate by the RRD aggregation on Munin Graph.

Handily the `tensor.sources.munin.MuninNode` source takes care of this by caching the metric between run intervals when that type is used.

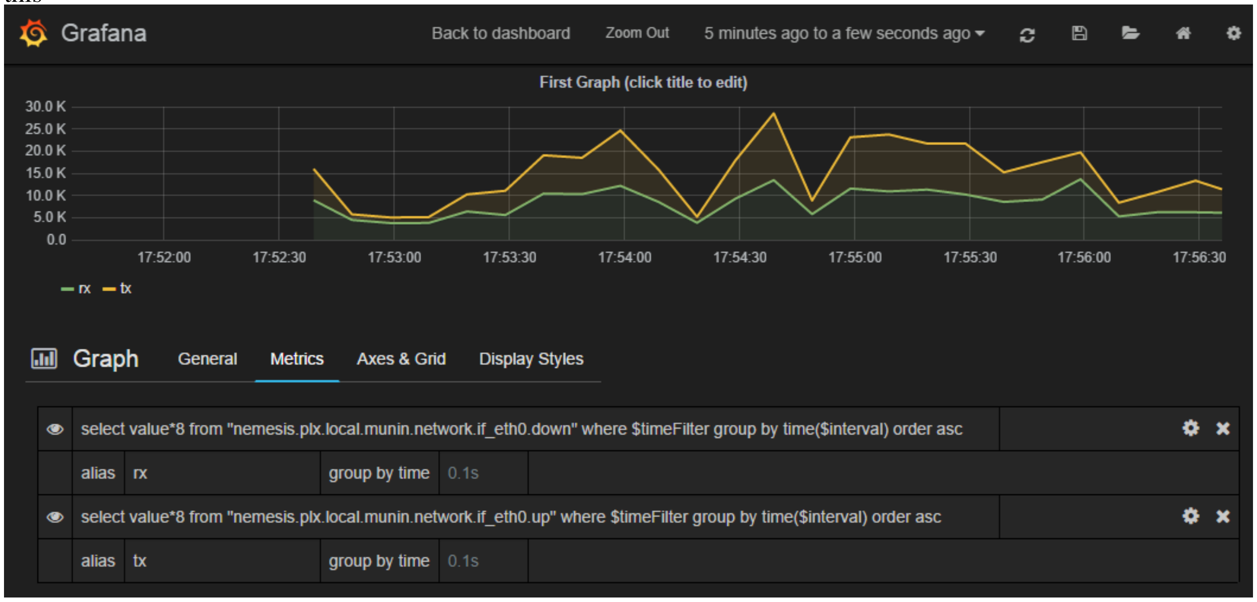
If we wanted to graph our network interface all we need to do is make it a slightly better unit by multiplying the Byte/sec metric by 8, since Grafana provides a bit/sec legend format.

To do this start by clicking the gear icon on the metric query, then select *Raw query mode*.

Use the following query

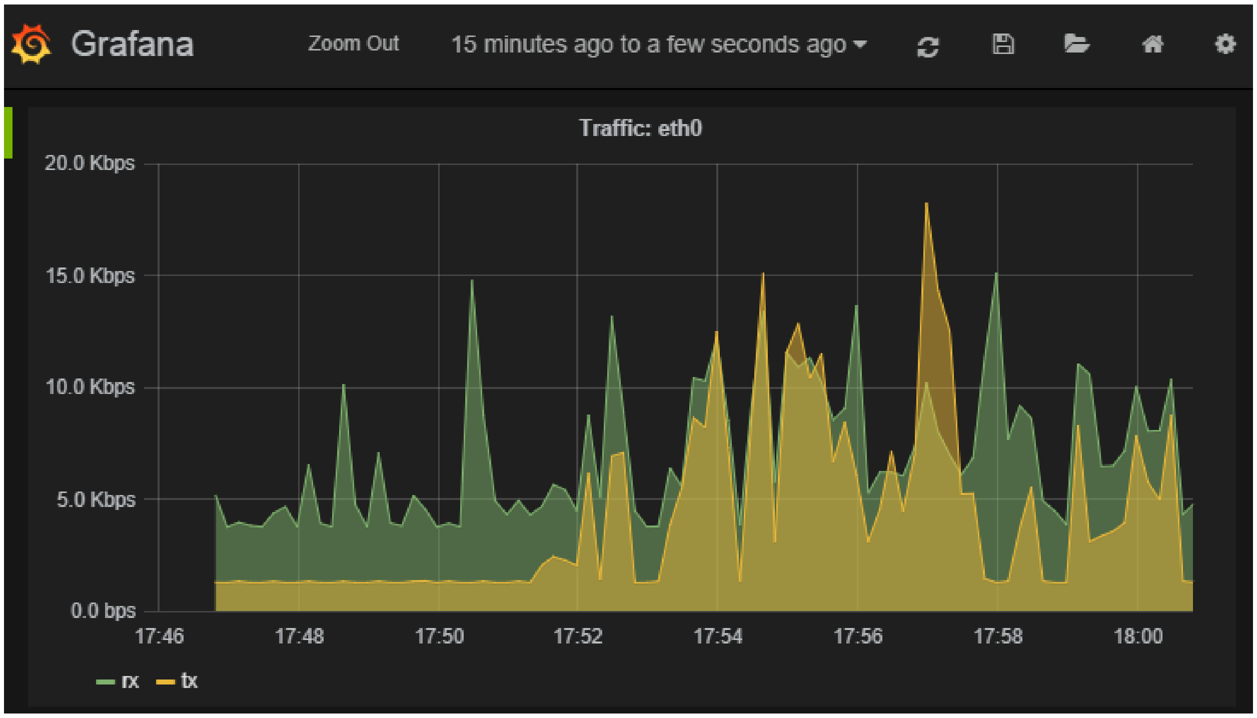
```
select value * 8 from "<your hostname>.munin.network.if_eth0.down" where $timeFilter group by time($:
```

And chose an alias of "RX". Do the same for if_eth0.up and alias that "TX". You should end up with something like this



```
select value*8 from "nemesis.plx.local.munin.network.if_eth0.down" where $timeFilter group by time($interval) order asc
alias  rx    group by time  0.1s
select value*8 from "nemesis.plx.local.munin.network.if_eth0.up" where $timeFilter group by time($interval) order asc
alias  tx    group by time  0.1s
```

Click on *General* to edit the title, and then on *Axes & Grid* change the Format to *bps*. Under *Display Styles* you can stack the data or play around with the look of the graph. Click *Back to dashboard* and you should end up with something as follows



API Documentation:

CHAPTER 5

tensor

5.1 tensor.interfaces

5.2 tensor.objects

class `tensor.objects.`**`Event`**(*state*, *service*, *description*, *metric*, *ttl*, *tags=[ ]*, *hostname=None*)

Bases: `object`

Tensor Event object

All sources pass these to the queue, which form a proxy object to create protobuf Event objects

Arguments:

State Some sort of string < 255 chars describing the state

Service The service name for this event

Description A description for the event, ie. “My house is on fire!”

Metric int or float metric for this event

Keyword arguments:

Tags List of tag strings

Hostname Hostname for the event (defaults to system fqdn)

class `tensor.objects.`**`Output`**(*config*, *tensor*)

Bases: `object`

Output parent class

Outputs can inherit this object which provides a construct for a working output

Arguments:

Config Dictionary config for this queue (usually read from the yaml configuration)

Tensor A TensorService object for interacting with the queue manager

`createClient`()

Deferred which sets up the output

`eventsReceived`()

Receives a list of events and processes them

Arguments: events – list of *tensor.objects.Event*

stop()
Called when the service shuts down

class tensor.objects.**Source**(*config*, *queueBack*, *tensor*)
Bases: object

Source parent class

Sources can inherit this object which provides a number of utility methods.

Arguments:

Config Dictionary config for this queue (usually read from the yaml configuration)

QueueBack A callback method to recieve a list of Event objects

Tensor A TensorService object for interacting with the queue manager

createEvent(*state*, *description*, *metric*, *prefix=None*)
Creates an Event object from the Source configuration

startTimer()
Starts the timer for this source

tick(**args*, ***kwargs*)
Called for every timer tick. Calls self.get which can be a deferred and passes that result back to the queueBack method

Returns a deferred

5.3 tensor.service

class tensor.service.**TensorService**(*config*)
Bases: twisted.application.service.Service

Tensor service

Runs timers, configures sources and and manages the queue

sendEvent(*event*)
Callback that all event sources call when they have a new event or list of events

setupOutputs(*config*)
Setup output processors

setupSources(*config*)
Sets up source objects from the given config

5.4 tensor.utils

class tensor.utils.**BodyReceiver**(*finished*)
Bases: twisted.internet.protocol.Protocol

Simple buffering consumer for body objects

class tensor.utils.**ProcessProtocol**(*deferred*, *timeout*)
Bases: twisted.internet.protocol.ProcessProtocol

ProcessProtocol which supports timeouts

`tensor.utils.`**`fork`**(*executable*, *args=()*, *env={}*, *path=None*, *timeout=3600*)

Provides a deferred wrapper function with a timeout function

Arguments:

Executable Executable

Keyword arguments:

Args Tupple of arguments

Env Environment dictionary

Timeout Kill the child process if timeout is exceeded

CHAPTER 6

tensor.protocol

6.1 tensor.protocol.riemann

class `tensor.protocol.riemann.`**`RiemannClientFactory`**

Bases: `twisted.internet.protocol.ReconnectingClientFactory`

A reconnecting client factory which creates RiemannProtocol instances

class `tensor.protocol.riemann.`**`RiemannProtocol`**

Bases: `twisted.protocols.basic.Int32StringReceiver`, `tensor.protocol.riemann.RiemannProtobu`

Riemann protobuf protocol

class `tensor.protocol.riemann.`**`RiemannUDP`**(*host*, *port*)

Bases: `twisted.internet.protocol.DatagramProtocol`, `tensor.protocol.riemann.RiemannProtobuf`

UDP datagram protocol for Riemann

CHAPTER 7

tensor.sources

7.1 tensor.sources.network

class `tensor.sources.network.`**`HTTP`**`(`*config*, *queueBack*, *tensor*`)`

Bases: `tensor.objects.Source`

Performs an HTTP request

Configuration arguments:

Method HTTP request method to use

Match A text string to match in the document when it is correct

Useragent User-Agent header to use

Metrics:

(service name).latency Time to complete request

class `tensor.sources.network.`**`Ping`**`(`*config*, *queueBack*, *tensor*`)`

Bases: `tensor.objects.Source`

Performs an Ping checks against a destination

Configuration arguments:

Destination Host name or IP address to ping

Metrics:

(service name).latency Ping latency

(service name).loss Packet loss

You can also override the *hostname* argument to make it match metrics from that host.

7.2 tensor.sources.munin

class `tensor.sources.munin.`**`MuninNode`**`(`**a*, ***kw*`)`

Bases: `tensor.objects.Source`

Connects to munin-node and retrieves all metrics

Configuration arguments:

Host munin-node hostname (probably localhost)

Port munin-node port (probably 4949)

Metrics:

(service name).(plugin name).(keys...) A dot separated tree of munin plugin keys

class `tensor.sources.munin.`**`MuninProtocol`**
Bases: `twisted.protocols.basic.LineReceiver`

MuninProtocol - provides a line receiver protocol for making requests to munin-node

Requests must be made sequentially

7.3 tensor.sources.linux

7.3.1 tensor.sources.linux.basic

class `tensor.sources.linux.basic.`**`CPU`**`(`**a*`)`
Bases: `tensor.objects.Source`

Reports system CPU utilisation as a percentage/100

Metrics:

(service name) Percentage CPU utilisation

class `tensor.sources.linux.basic.`**`DiskFree`**`(`*config*, *queueBack*, *tensor*`)`
Bases: `tensor.objects.Source`

Returns the free space for all mounted filesystems

Metrics:

(service name).(device) Used space (%)

class `tensor.sources.linux.basic.`**`LoadAverage`**`(`*config*, *queueBack*, *tensor*`)`
Bases: `tensor.objects.Source`

Reports system load average for the current host

Metrics:

(service name) Load average

class `tensor.sources.linux.basic.`**`Memory`**`(`*config*, *queueBack*, *tensor*`)`
Bases: `tensor.objects.Source`

Reports system memory utilisation as a percentage/100

Metrics:

(service name) Percentage memory utilisation

7.3.2 tensor.sources.linux.process

class `tensor.sources.linux.process.`**`ProcessCount`**`(`*config*, *queueBack*, *tensor*`)`
Bases: `tensor.objects.Source`

Returns the ps count on the system

Metrics:

(service name) Number of processes

7.4 tensor.sources.media

7.4.1 tensor.sources.media.libav

class `tensor.sources.media.libav.`**`DarwinRTSP`**(*config*, *queueBack*, *tensor*)

Bases: `tensor.objects.Source`

Makes avprobe requests of a Darwin RTSP sample stream (sample_100kbit.mp4)

Configuration arguments:

Destination Host name or IP address to check

Metrics: :(service name): Time to complete request

You can also override the *hostname* argument to make it match metrics from that host.

CHAPTER 8

tensor.outputs

8.1 tensor.outputs.riemann

CHAPTER 9

Indices and tables

- *genindex*
- *modindex*
- *search*