# cubism.js - xiaofenguo - 博客园

http://www.cnblogs.com/xiaofenguo/p/6595054.html

Cubism.js 是时间序列化的一个D3插件，使用Cubism构建更好的实时指示板，从Graphite,Cube 和其他的资源中拉拉取数据。在GitHub的Apache License上可以获取Cubism

可称量的（scalable）

Cubism 逐步地获取时间序列数据：在初始化显示之后，Cubsim通过只获取最近的数据减少服务器的压力。Cubism 逐步渲染，通过canvas将图表向左边移动一个像素，这种方法允许Cubism很容易的扩展到每10秒钟刚更新数以百计的度量。尽管是异步获取数据，同步渲染使图表的更新的同步的，进一步提高了性能和可读性。

有效的 （Effective）

Cubism 同样尺度的感应:小倍数对齐时间比较方便快捷，Cubism的水平图表比标准图表更好的利用垂直空间........待续

分类: [D3](http://www.cnblogs.com/xiaofenguo/category/969692.html)

# Cubism.js

http://square.github.io/cubism/

Time Series Visualization

04:5905 PM05:0105:0205:0305:0405:0505:0605:0705:0805:0905:1005:1105:1205:1305:14

foo9.7

bar1.4

foo + bar11

foo - bar8.4

Mouseover or use the arrow keys to inspect values.  
[Open in a new window.](http://square.github.io/cubism/demo/)

**Cubism.js** is a [D3](http://mbostock.github.com/d3/) plugin for visualizing time series. Use Cubism to construct better realtime dashboards, pulling data from [Graphite](http://graphite.wikidot.com/), [Cube](http://square.github.io/cube/) and other sources. Cubism is available under the [Apache License](http://www.apache.org/licenses/LICENSE-2.0.html) on [GitHub](https://github.com/square/cubism).

**Scalable**

Cubism fetches time series data incrementally: after the initial display, Cubism reduces server load by polling only the most recent values. Cubism renders incrementally, too, using [Canvas](http://www.w3.org/TR/2dcontext/) to shift charts one pixel to the left. This approach lets Cubism scale easily to hundreds of metrics updating every ten seconds! Despite asynchronous fetching, rendering is synchronized so that charts update simultaneously, further improving performance and readability.

**Effective**

Cubism also scales in terms of perception: small multiples aligned by time facilitate rapid comparison. Cubism’s horizon charts make better use of vertical space than standard area charts, allowing you to see many more metrics at-a-glance and increasing the likelihood of discovery. If you compress a 120-pixel tall area chart to 30 pixels, you lose 75% of the resolution and it becomes harder to see small changes:

Area (120px)9.7

Area (30px)9.7

In contrast, [horizon charts](http://vis.berkeley.edu/papers/horizon/) reduce vertical space *without* losing resolution. Larger values are overplotted in successively darker colors, while negative values are offset to descend from the top. As you increase the number of colors, you reduce the required vertical space:

Horizon, 1-band (120px)9.7

Horizon, 2-band (60px)9.7

Horizon, 3-band (40px)9.7

Horizon, 4-band (30px)9.7

By combining position and color, horizon charts [improve perception](http://www.perceptualedge.com/articles/visual_business_intelligence/time_on_the_horizon.pdf): position is highly effective at discriminating small changes, while color differentiates large changes. To further increase data density, Cubism favors per-pixel metrics where each pixel encodes a distinct point in time. Cubism also includes thoughtful default colors by [Cynthia Brewer](http://colorbrewer2.org/).

**Flexible**

Cubism is data-source agnostic. It has built-in support for [Graphite](https://github.com/square/cubism/wiki/Graphite) and [Cube](https://github.com/square/cubism/wiki/Cube), and can be readily extended to fetch data from other sources. Client-side [metric arithmetic](https://github.com/square/cubism/wiki/Metric) allows further flexibility by combining metrics from multiple sources. Cubism’s modular components are designed for extensibility. You can add new chart types and modes of interaction, too. Cubism builds on [D3](http://mbostock.github.com/d3/), making it highly customizable via CSS and JavaScript.

Want to learn more? [See the source and documentation.](https://github.com/square/cubism)

# square/cubism:

Cubism.js: A JavaScript library for time series visualization.

https://github.com/square/cubism

# 时序js插件cubism使用 - 骡骡 - 博客园

http://www.cnblogs.com/liujitao79/p/4685197.html

document

<http://iwantmyreal.name/blog/2012/09/16/visualising-conair-data-with-cubism-dot-js>  
<https://hveem.no/vm-monitoring-using-salt-and-cubism>  
<http://sciviz.info/time-series-visualization-using-cubism-js>  
<http://square.github.io/cubism/>

api

<https://github.com/square/cubism/wiki>

example

<script>

function random(name) {

var value = 0,

values = [],

i = 0,

last;

/\* 支持cube, graphite内建方法, 也可以使用context.metric()方法构造新的数据源。

context.metric()有一个request函数参数, 还有一个可选的name参数。start & stop是日期类型，step单位ms。

callback有2个参数，第1参数用于node.js，第2参数是数值形数组 \*/

return context.metric(function(start, stop, step, callback) {

start = +start, stop = +stop;

if (isNaN(last)) last = start;

while (last < stop) {

last += step;

value = Math.max(-10, Math.min(10, value + .8 \* Math.random() - .4 + .2 \* Math.cos(i += .2)));

values.push(value);

}

callback(null, values = values.slice((start - stop) / step));

}, name);

}

var context = cubism.context()

.serverDelay(0) // 服务器延时

.clientDelay(0) // 客户端延时

.step(1e3) // 步进1e3（1000ms）, cubism的时间单位是ms

.size(960); // 960 \* 1s, 时间轴是16分钟

var foo = random("foo"),

bar = random("bar");

// example1

d3.select("#example1").call(function(div) {

div.append("div")

.attr("class", "axis")

.call(context.axis().orient("top"));

div.selectAll(".horizon")

.data([foo, bar, foo.add(bar), foo.subtract(bar)])

.enter().append("div")

.attr("class", "horizon")

.call(context.horizon().extent([-20, 20]));

div.append("div")

.attr("class", "rule")

.call(context.rule());

});

// example2a

d3.select("#example2a").call(function(div) {

div.datum(foo);

div.append("div")

.attr("class", "horizon")

.call(context.horizon()

.height(120)

.mode("mirror")

.colors(["#bdd7e7","#bae4b3"])

.title("Area (120px)")

.extent([-10, 10]));

div.append("div")

.attr("class", "horizon")

.call(context.horizon()

.height(30)

.mode("mirror")

.colors(["#bdd7e7","#bae4b3"])

.title("Area (30px)")

.extent([-10, 10]));

});

// example2b

d3.select("#example2b").call(function(div) {

div.datum(foo);

div.append("div")

.attr("class", "horizon")

.call(context.horizon()

.height(120)

.colors(["#bdd7e7","#bae4b3"])

.title("Horizon, 1-band (120px)")

.extent([-10, 10]));

div.append("div")

.attr("class", "horizon")

.call(context.horizon()

.height(60)

.colors(["#6baed6","#bdd7e7","#bae4b3","#74c476"])

.title("Horizon, 2-band (60px)")

.extent([-10, 10]));

div.append("div")

.attr("class", "horizon")

.call(context.horizon()

.height(40)

.colors(["#3182bd","#6baed6","#bdd7e7","#bae4b3","#74c476","#31a354"])

.title("Horizon, 3-band (40px)")

.extent([-10, 10]));

div.append("div")

.attr("class", "horizon")

.call(context.horizon()

.height(30)

.colors(["#08519c","#3182bd","#6baed6","#bdd7e7","#bae4b3","#74c476","#31a354","#006d2c"])

.title("Horizon, 4-band (30px)")

.extent([-10, 10]));

});

// On mousemove, reposition the chart values to match the rule.

context.on("focus", function(i) {

d3.selectAll(".value").style("right", i == null ? null : context.size() - i + "px");

});

我的程序

<html>

<head>

<title>dashboard</title>

<script src="{{ url\_for('static', filename='js/d3.min.js') }}"></script>

<script src="{{ url\_for('static', filename='js/cubism.v1.min.js') }}"></script>

<script src="{{ url\_for('static', filename='js/jquery-1.11.3.min.js') }}"></script>

<link rel="stylesheet" type="text/css" href="{{ url\_for('static', filename='css/style.css') }}">

</head>

<body>

<div id="cubism"></div>

</body>

<script type="text/javascript">

function analysis(name, cname, bandwidth, interface, threshold, measure) {

var label = ((measure.split('\_')[1] == 'rx') ? 'In' : 'Out') + ' ' +cname;

var result = context.metric(function(start, stop, step, callback) {

$.ajax({

type: 'GET',

url: '/api',

data: {

'name': name,

'interface': interface,

'measure': measure,

'start': start.toISOString(),

'stop': stop.toISOString()

},

dataType: 'json',

success: function(data) {

callback(null, data);

//console.log(data);

},

});

}, label);

return result;

}

var seconds = 60;

var default\_colors = ['#08519c', '#3182bd', '#6baed6', '#bdd7e7', '#bae4b3', '#74c476', '#31a354', '#006d2c'];

var custom\_colors = ['#ffcc99', '#ff9999', '#ffcc00', '#ff6666'];

//default\_colors = custom\_colors;

var context = cubism.context()

.serverDelay(seconds\*1000)

.step(seconds\*1000)

.size(60\*16);

var dataset = [];

{% for switch in switches %}

var info = ['{{ switch.name }}', '{{ switch.cname }}', '{{ switch.bandwidth }}', '{{ switch.interface }}', '{{ switch.threshold }}'];

var traffic\_in = analysis(info[0], info[1], info[2], info[3], info[4], 'snmp\_rx'),

traffic\_out = analysis(info[0], info[1], info[2], info[3], info[4], 'snmp\_tx');

dataset.push(traffic\_in, traffic\_out);

{% endfor %}

d3.select("#cubism").call(function(div) {

div.append("div")

.attr("class", "axis")

.call(context.axis()

.tickFormat(d3.time.format("%H:%M"))

.ticks(d3.time.minutes, 60)

.orient("top"));

div.selectAll(".horizon")

.data(dataset)

.enter().append("div")

.attr("class", "horizon")

.call(context.horizon()

.height(30)

.colors(default\_colors)

.format(d3.format('.2f')));

div.append("div")

.attr("class", "rule")

.call(context.rule());

});

context.on("focus", function(i) {

d3.selectAll(".value").style("right", i == null ? null : context.size() - i + "px");

});

</script>

</html>

#!/usr/bin/env python

# -\*- encoding: utf-8 -\*-

import json

import yaml

from flask import Flask, render\_template, request, url\_for

from influxdb import InfluxDBClient

app = Flask(\_\_name\_\_)

client = InfluxDBClient(host='10.10.32.109', port=8086, username='root', password='root', database='collectd')

@app.route('/', methods=['GET'])

def index():

file = 'switch.yaml'

with open(file) as f:

switches = yaml.load(f)

return render\_template('dashboard.html', switches=switches)

@app.route('/cubism', methods=['GET'])

def cubism():

return render\_template('cubism.html')

@app.route('/api', methods=['GET'])

def api():

host, interface, measure, start, stop = \

request.args['name'], request.args['interface'], request.args['measure'], \

request.args['start'], request.args['stop']

file = 'dashboard.yaml'

query = 'select derivative(value)\*8/1000/1000 from '+ measure \

+ ' where host=\'' + host + '\' and type\_instance=\'' + interface \

+ '\' and time > \'' + start \

+ '\' and time < \'' + stop +'\''

print query

result = client.query(query)

return json.dumps([value[1] for value in result.raw['series'][0]['values'] ])

if \_\_name\_\_ == '\_\_main\_\_':

app.run(host='0.0.0.0', debug=True)

# influxdb的sql可能有bug，按时间段取值，结果会丢失第1个值

# curl http://127.0.0.1:5000/api?name=bjtn&measure=traffic\_out&start=2015-07-28T08:10:00Z&stop=2015-07-28T08:50:59Z

标签: [js](http://www.cnblogs.com/liujitao79/tag/js/)

# Visualising ConAir data with Cubism.js

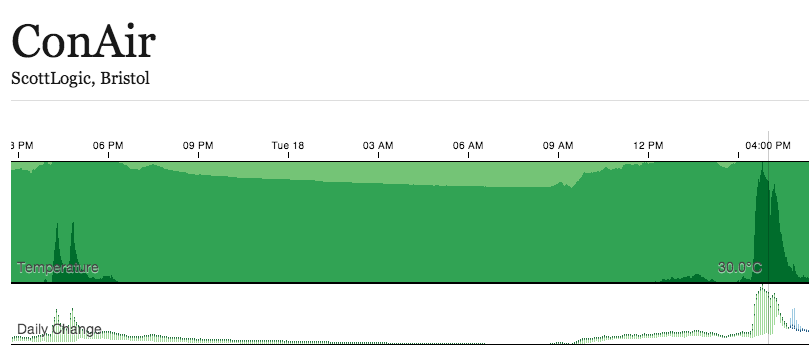
http://iwantmyreal.name/blog/2012/09/16/visualising-conair-data-with-cubism-dot-js

This post is part of a series of blog posts detailing how I built a system which records the temperature in our office. The parts are as follows:

* [Measuring temperature with an Arduino](http://iwantmyreal.name/blog/2012/09/23/measuring-the-temperature-with-an-arduino-and-a-thermistor) describes how to use an Arduino and a thermistor to record the temperature.
* [ConAir](http://iwantmyreal.name/blog/2012/09/14/conair-the-quest-for-reasonable-office-air-con) gives an overview of the motivation behind the project, and runs through the process of taking the readings from the arduino and publishes them to a TempoDB database on the internet.
* This post describes building a proxy webservice for the TempoDB database, and then visualising the temperature data using cubism.js
* [iOS App](http://iwantmyreal.name/blog/2012/12/11/an-ios-app-for-plotting-live-data-conair-ios) describes building an iOS app which pulls the data from the webservice created in this app and uses ShinobiCharts to plot the result.

Hot on the tail of being able to record temperature readings from the Arduino in the office, we can get some charting on the go.

This post describes building a [cubism.js](http://square.github.com/cubism) front end to a [TempoDB](http://tempo-db.com/) API proxy we implement in Sinatra. You can see the result running live on heroku at [sl-conair](http://sl-conair.herokuapp.com/), and there’s a screenshot below in case we’re working on the electronics and there is no data:



We have been using [TempoDB](http://tempo-db.com/) to store the temperature data points - it has a great API for querying your dataset, including rollups, which are able to summarise your data at a resolution of your choice. Unfortunately, TempoDB doesn’t yet allow public access to datasets - the API requires authentication for both read and write. Therefore the first part of this stage will be to build a proxy for the TempoDB API. We then use Cubism.js to interface this proxied API.

TempoDB Proxy

TempoDB provide a selection of API clients - we used the python one to upload the data points as they are read off the Arduino. Here I’m going to use the Ruby one - just ‘cos.

The following is part of a really simple Sinatra application which will interface with the TempoDB API.

We receive requests for data on /data with URL parameters start, stop and step. The two timestamps are in a format which can be parsed by ruby’s Time.parse method, whilst the step is measured in milliseconds. This is to fit nicely with Cubism.js.

We convert the step into a string of a suitable format for the TempoDB client, here, assuming that the step will always be an integer number of minute. We then send the request to TempoDB.

As mentioned before, if you ask for data at a coarser level than that at which it was recorded, TempoDB will “roll” it up for you. This is a fantastic feature, and although the logic to perform this kind of operation isn’t complex, this kind of operation can absorb days worth of optimisation time, combined with the fiddling associated with working with date objects. Here, we are telling TempoDB that the rollup function should be mean, although other sensible collection operators are implemented too (e.g. count, sum, max etc).

Discontinuous Data

If there is a discontinuity in your data, then TempoDB won’t perform any interpolation - there will be gaps in what it returns. This is fine because the data it returns consists of timestamp-value pairs.

This does however present an issue with Cubism.js, which doesn’t expect discontinuities in the data, and in fact only works with an array of values, ignoring timestamps.

Therefore we iterate through the array of datapoints returned by TempoDB and if there is a temporal discontinuity, we interpolate suitable values. In this instance we’ve chosen to linearly interpolate between the points either side of the discontinuity.

**if**((next\_time **-** current\_time **-** step **/** 1000).**abs** **>** 5)

*# Let's add the right number of values*

points\_needed **=** ((next\_time **-** current\_time) **/** (step **/** 1000)).**floor**

difference **=** data[index**+**1].**value** **-** val.**value**

points\_needed.**times** { **|**i**|** response\_data.**push**({ value: (val.**value** **+** difference **\*** i **/** points\_needed.**to\_f**) }) }

**end**

This particular endpoint will return a JSON array of hashes, each containing a value key. The array will be in time order, and the elements represent consecutive temperature readings, at the specified time interval, from start to stop.

Cubism.js

Cubism is a time-series visualisation tool built on top of the brilliant [d3.js](http://d3js.org/)javascript library. d3.js works on the paradigm of data driven websites - where the content and the style changes as events occur within the data - whether it be a user interacting with it, or new data points arriving. There are some awesome d3.js demos on their website with fantastic visualisations - easily a way to waste an hour…

[Cubism.js](http://square.github.com/cubism) is a library developed by the people at square for displaying just this kind of data. We are going to use it to call our new API proxy.

Start with a div, within which we will place the chart:

<**div** id="chart"></**div**>

Cubism has the concept of a context, which manages the data requests and the UI elements.

**var** context**=** cubism.context()

.serverDelay(2 **\*** 60 **\*** 1000) *// Allow 2 mins server delay*

.step(2 **\*** 60 **\*** 1000) *// Every 2 mins*

.size(940)

The serverDelay specifies how long a delay we are prepared to wait before querying the server for new data points, step defines how many milliseconds you wish to wait between datapoints and size determines how many datapoints (and therefore the width in pixels) you wish the chart to display.

Cubism manages the requests for the data points given the above settings, and a data source. It has built-in datasource types for [Cube](http://square.github.com/cube) and [Graphite](http://graphite.wikidot.com/), but we need to create our own:

**var** primary **=** temperature(),

esecondary **=** primary.shift(**-** 24 **\*** 60 **\*** 60 **\*** 1000);

**function** **temperature**() {

**return** context.metric(**function**(start, stop, step, callback) {

d3.json("/data/?start=" **+** start.toISOString()

**+** "&stop=" **+** stop.toISOString()

**+** "&step=" **+** step, **function**(data) {

**if**(**!**data) **return** callback(**new** Error("unable to load data"));

callback(**null**, data.map(**function**(d) { **return** d.value; }));

});

});

}

We have defined 2 metrics - the second of them simply a time-shifted version of the first. This is the simple kind of metric calculations provided by cubism, and we’ll use this in the difference chart.

The temperature() function returns a context metric, which describes how cubism should request data given start, stop and step values, and then returns the data with a node.js inspired callback method. It’s all pretty straightforward - we just plug in our particular API URL and then make sure we process the results to pass an array of numerical values to the callback function.

That’s all the data handling work done. Now we just need to sort the GUI.

d3.select("#chart").call(**function**(div) {

div.append("div")

.attr("class", "axis")

.call(context.axis().orient("top"));

div.selectAll(".horizon")

.data([primary])

.enter().append("div")

.attr("class", "horizon")

.call(context.horizon()

.height(120)

.format(d3.format(".2f"))

.title("Temperature"));

div.selectAll(".comparison")

.data([[primary, secondary]])

.enter().append("div")

.attr("class", "comparison")

.call(context.comparison()

.height(60)

.formatChange(d3.format(".1f%"))

.title("Daily Change"));

div.append("div")

.attr("class", "rule")

.call(context.rule());

});

context.on("focus", **function**(i) {

format **=** d3.format(".1f");

d3.selectAll(".horizon .value").style("right", i**==** **null** ? **null** : context.size() **-** i **+** "px")

.text(format(primary.valueAt(Math.floor(i))) **+** "\u00B0C");

});

This code first find the div we defined beforehand, adds an axis to the top of it, adds a horizon chart and a comparison chart and a rule which follows the cursor over the map.

The last block updates the value display to follow the rule.

Lots more info on setting this up is available on the cubism site, and there are plenty of demos to hack around with.

Et voila…

After all that work, you probably want to see it. You can check the current graph at the site I’ve popped up on heroku [sl-conair](http://sl-conair.herokuapp.com/). Be warned that this project is a work in progress - we’ve got loads we want to do with the electronics, the data collection and the frontend, so it’s quite likely that there will be gaps in the data.

Hope that was of interest. I still want to post about the electronics side of this project - hopefully I’ll get round to that soon

sx

Edit (11/02/2013)

Added an additional introduction to pull together the ConAir posts.

# Vm monitoring using salt and cubism

https://hveem.no/vm-monitoring-using-salt-and-cubism

Monitoring is not easy. Fetching metrics and creating graphs is easy. But extracing insight from graphs is quite the other task.

I have been investigating different routes. There's a phletora of solutions all solving different subsets of the problem. As always, I'm looking for solutions that are modular with orthogonal components. That helps the different pieces to do one thing, and do one thing **well**. During my adventure in the monitoring jungle I came across [Cubism](http://square.github.com/cubism/). Cubism is a time series visualization plugin in javascript on top of the [D3 js library](http://d3js.org/). The best thing about Cubism is how it is able to display alot of data in a really compact manner, by using [horizon charts](http://vis.berkeley.edu/papers/horizon/). I got really inspired by watching Mike Bostock explain it all in [this video](http://vimeo.com/42176902) ([slides](http://bost.ocks.org/mike/cubism/intro/)) and promptly decided to try and build my own simple virtual machine monitoring inside my little [saltvirt](https://hveem.no/saltvirt) project.

The way cubism works is requesting historic data for a time range, and then polling frequently the backend for fresh data to graph. In the setup I'm about to describe, there's no backend that stores the data I want to graph, so when the user opens up the monitoring it will start out blank and then start populating. For this proof of concept that's fine, since it enables me to graph data without introducing any new compononents into my existing environment. If I wanted this into production, I would use [graphite](http://graphite.wikidot.com/), [cube](https://github.com/square/cube/wiki) or even write a salt returner into mongodb for storing the time series data.

The Salt module function to poll data

I added a utility function to the existing Salt module virt.py to fetch CPU usage. It supports returning all VMs or a single named VM:

salt/modules/virt.py

def vm\_cputime(vm\_=None):

'''

Return cputime used by the vms on this hyper in a

list of dicts::

[

'your-vm': {

'cputime' <int>

'cputime\_percent' <float>

},

...

]

If you pass a VM name in as an argument then it will return info

for just the named VM, otherwise it will return all VMs.

CLI Example::

salt '\*' virt.vm\_cputime

'''

host\_cpus = \_\_get\_conn().getInfo()[2]

def \_info(vm\_):

dom = \_get\_dom(vm\_)

raw = dom.info()

cputime = int(raw[4])

cputime\_percent = 0

if cputime:

cputime\_percent = 1.0e-7 \* cputime / host\_cpus

return {

'cputime': int(raw[4]),

'cputime\_percent': '%.0f' %cputime\_percent

}

info = {}

if vm\_:

info[vm\_] = \_info(vm\_)

else:

for vm\_ in list\_vms():

info[vm\_] = \_info(vm\_)

return info

**The HTML**

Include the requirements and specify the graph container.

index.html

<script src="/static/d3.v3.min.js"></script>

<script src="/static/cubism.v1.min.js"></script>

<div id="graphs"></div>

**The JavaScript**

As I noted earlier, this is a proof of concept, and its biggest weak spot is in the javascript below. I feed one metric to cubism per VM. Each metric has a callback that runs 1 salt call to its corresponding VM. So in effect you have 1 AJAX call per VM per second, which translates into 1 salt command per second. This will make salt API, salt and the libvirtd processes very busy and it will even impact the performance on each of these systems. To improve this, one would have to change this to make a javascript cubism backend that has 1 ajax call for ALL the VMs, and then split the results into proper arrays that cubism can read. I didn't spend time fixing this because I think the road ahead is using a proper database backend anyway.

saltvirt.js

*// Create cubism context*

context = cubism.context()

.serverDelay(10 \* 1000) *// allow 10 seconds of collection lag*

.step(1 \* 1000) *// 1 seconds per value*

.size(1080); *// fetch 1080 values (pixels)*

*/\* Custom metric that polls Salt \*/*

var cputime = function(host, vm) {

var value = 0,

values = [],

realvalues = [],

i = 0,

last;

return context.metric(function(start, stop, step, callback) {

start = +start, stop = +stop;

if (isNaN(last)) last = start;

$.post(api\_url\_base + '/', tcontext({

tgt:host,

fun:'virt.vm\_cputime',

arg:[vm]

}), function(data) {

if(data['return'] != undefined) {

var ret = data['return'][0][host];

var cputime\_percent = parseInt(ret[vm].cputime\_percent);

var lastval = realvalues.slice(-1)[0];

*// Need a real value to calculate diff*

if (lastval > 0) {

var thisval = cputime\_percent - lastval;

values.push(thisval);

} else {

values.push(0);

}

realvalues.push(cputime\_percent);

}else {

*// Libvirt no history :(*

*// Fill values with 0*

while (last < stop) {

last += step;

value = 0;

values.push(value);

realvalues.push(value);

}

}

values = values.slice((start - stop) / step);

realvalues = realvalues.slice((start - stop) / step);

callback(null, values );

});

}, String(vm).replace(sv.strip\_domain,''));

}

*// Fetch all the running VMs on all hosts and create metrics for each*

$.post(api\_url\_base + '/', tcontext({

tgt:'\*', *// All minions*

timeout: 30, *// wait a bit to get everything*

fun:'virt.list\_active\_vms' *// only fetch active vms*

}), function(data) {

var metrics = []; *// list of metrics*

$.each(data['return'], function(oidx, list) {

$.each(list, function(host, vmlist) {

*// Non-virt-hosts returns "not available"*

if(jQuery.type(vmlist) == jQuery.type([])) {

$.each(vmlist, function(vmidx, vm) {

*// Create a metric and add it to the list of metris*

metrics.push(cputime(host, vm));

});

}

});

});

*// Create the graphs and add to #graphs container*

d3.select("#graphs").call( function(div) {

*// An axis at the top*

div.append("div")

.attr("class", "axis")

.call(context.axis().orient("top"));

*// All the horizon graphs*

div.selectAll(".horizon")

.data(metrics)

.enter().append("div")

.attr("class", "horizon")

.call(context.horizon().extent([0, 100]));

div.append("div")

.attr("class", "rule")

.call(context.rule());

});

});

*// On mousemove, reposition the chart values to match the rule.*

context.on("focus", function(i) {

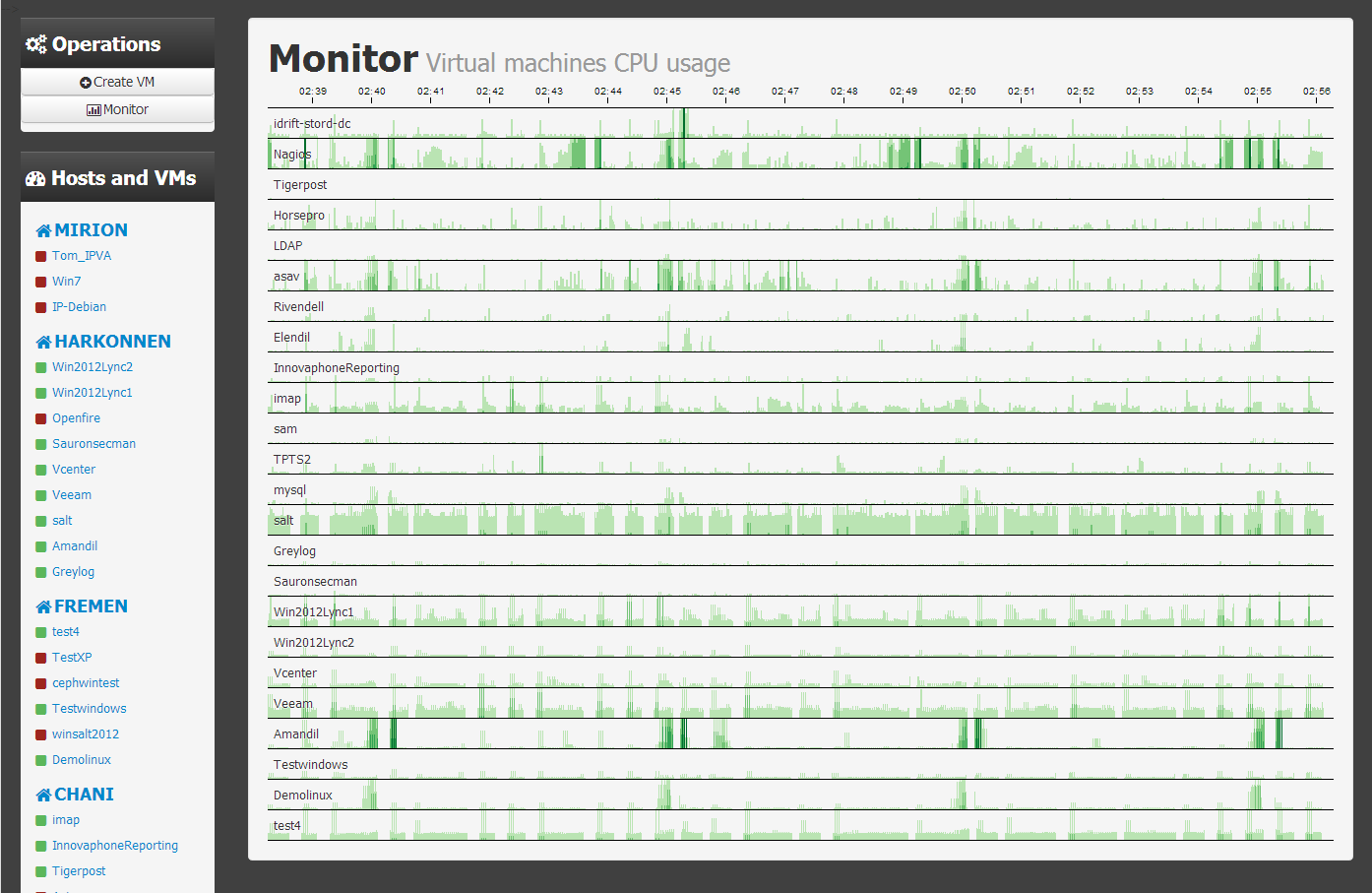
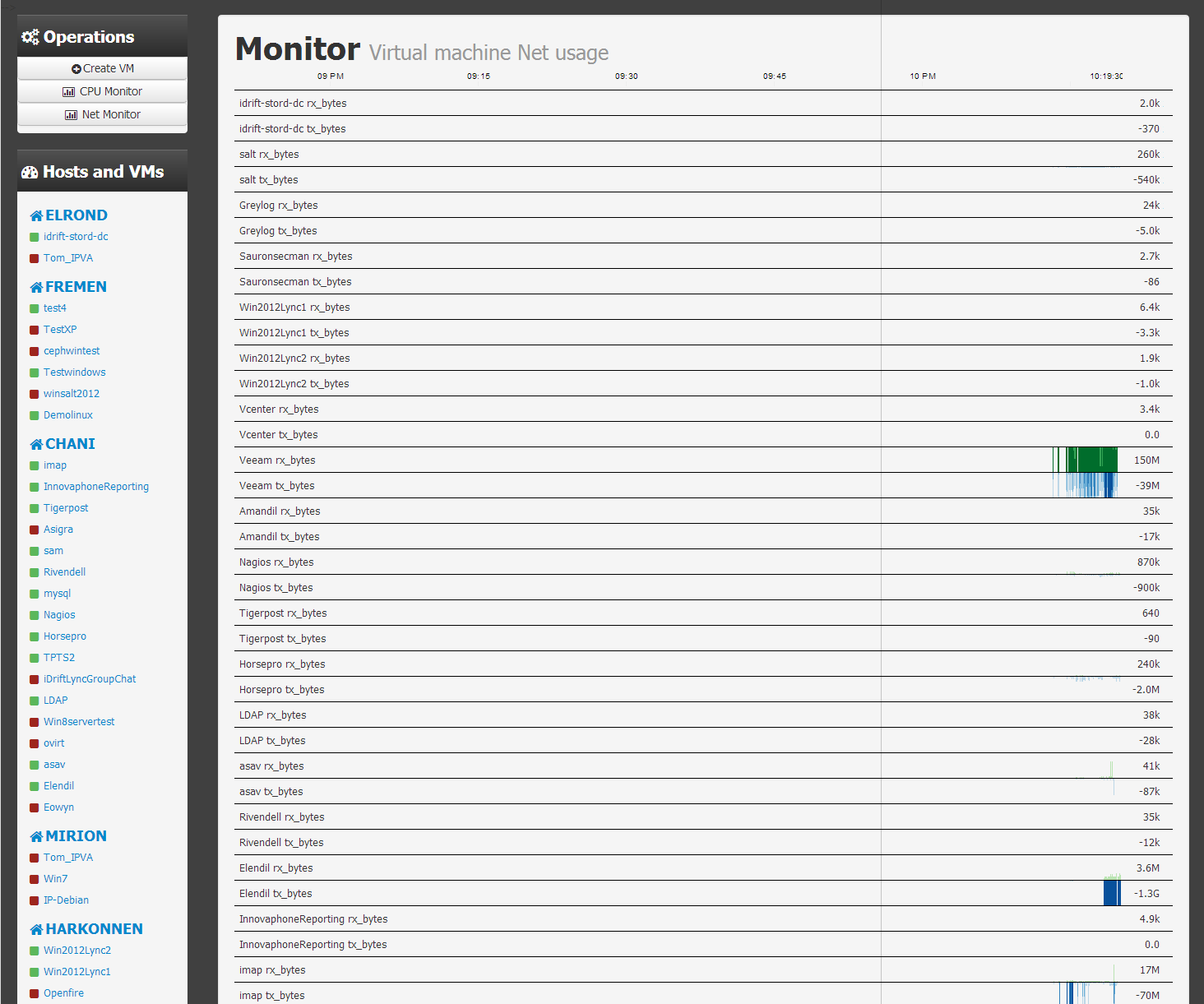
d3.selectAll(".value").style("right", i == null ? null : context.size() - i + "px");

});

});

**Screenshot**

As you can see, most of my VMs in my test cluster are fairly idle. The default in cubism is 4 bands, so when green area is at full height of its 30 pixels, its corresponding CPU usage percentage is 25%. There is some choppiness in the graphs because I dont do any time interpolation and when some of the data does not return in a timely manner it will "queue" and get displayed as a spike. Several ways to fix this, either by not ever getting lost data, or interpolate in js. But again, a proper database backend will deal with this.

CPU UsageNet Usage

Update: Check out[*Part #2*](https://hveem.no/salt-returner-for-carbon)for getting stateful graphing

**ALSO READ**

* [Visualizing Latency Variance With Grafana](https://hveem.no/visualizing-latency-variance-with-grafana)**22 MAY 2015**
* [Hacking Cubesensors](https://hveem.no/hacking-cubesensors)**20 DECEMBER 2014**
* [Using Dnsmasq For Dhcpv6](https://hveem.no/using-dnsmasq-for-dhcpv6)**20 DECEMBER 2014**
* [A Modern IRC Experience](https://hveem.no/a-modern-IRC-experience)**17 FEBRUARY 2014**
* [Using Docker As Lua Nginx Appserver](https://hveem.no/using-docker-as-lua-nginx-appserver)**19 DECEMBER 2013**
* [Salt Cli Visualization Using Runner And Outputter](https://hveem.no/salt-cli-visualization-using-runner-and-outputter)**13 APRIL 2013**
* [My Lua Micro Web Framework](https://hveem.no/about) **10.11.2012**