Faculty of Science and Technology

MASTER'S THESIS

Study program/ Specialization: Master of Science in Computer Science	Spring semester, 2014 Open access
Writer: Roberto Martín Muñoz	…… (Writer's signature)
Faculty supervisor: Tomasz Wiktor Włodarczyk External supervisor(s):	
Thesis title: **Scalable and user friendly user interface for time-series analytics for OpenTSDB**	
Credits (ECTS): 30	
Key words: user interface, OpenTSDB, NodeJS, WebSockets, R, JavaScript, opentsdbnode, nodetsd	Pages: ……69…… + enclosure: … code on CD … Stavanger, …23/06/2014..... Date/year

Scalable and user friendly user interface for time-series analytics for OpenTSDB

Roberto Martín Muñoz

Faculty of Science and Technology

University of Stavanger

July 2014

Abstract

OpenTSDB is a fast and reliable database used worldwide. While it has numerous advantages, its current web user interface is simplistic and not interactive, wasting the time that takes to perform a specific task .

This thesis focuses on the implementation of a more reliable and interactive architecture using a Model, View, Controller architecture while considering visual analytics, NodeJS, websockets, Python, and R.

The nodejs server is proposed as a solution. It has four different built-in connectors that obtain and transform data from OpenTSDB. We will show a connector from OpenTSDB to the NodeJS server (nodetsdb), OpenTSDB directly with the client javascript (nodetsdb-client), OpenTSDB to Python, and OpenTSDB with R.

After implementing and testing all the connectors we discovered that the connector from OpenTSDB to NodeJS is the fastest one, retrieving one month of data points in less than sixty ms.

Acknowledgements

Foremost, I would like to express my gratitude to Prof. Chunming Rong and my supervisor, Dr. Tomasz Wiktor Wlodarczyk for their valuable comments and help.

My sincere thanks also goes to my friend Manuel Caballero Sánchez that helped with critique and sincere feedback.

This work could not be done without the support of my fiancée Tatiana Popovitchenko that helped enormously in editing and moral support.

Last but not least I would like to thank my family in Spain and friends that supported me during this time.

Roberto Martín Muñoz
University of Stavanger

Contents

1 Introduction

OpenTSDB is a scalable database built on top of HBase and specifically designed for managing time series data. This database is used worldwide, specifically at the University of Stavanger (UiS), to store and access time series data such as daily weather and data from sensors around the building. This is done as a part of the project *Self learning Energy Efficient builDings and open Spaces* (SEEDS) in collaboration with the European Union. The great advantage of this database is its efficiency in managing a large amount of data points.

The OpenTSDB database has a web interface in which one can fetch data represented in plots. While it is adept at managing data, the interface is basic and does not allow the analyst to manage or obtain more information of the data points. Ultimately, this results in wasted time for the analyst, as they must change the query over and over again.

To address the problem of the interface, we will provide a real dashboard based on the latest web standards that will allow an analyst to navigate through the data in a more interactive way and provide contextual information about the data. This will allow the analyst to detect patterns more efficiently and gather more knowledge from the raw data. We will implement an account manager with proper security for the system, leaving the main structure so future versions will allow the analyst to have their own personal account to store favorite plots, most used plots, favorite metrics, and custom alerts in datasets (like range restriction).

A NodeJS server will be implemented to serve the page to the clients and fetch the points through different connectors with different advantages and disadvantages.

The implementation will be modular and based on the Model View Controller pattern and the three layers (analytics, data management, and visualization) that visual

analytics must relay to meet analyst requirements. In order to help the community behind OpenTSDB, the main server and the different connectors will be published with an open source license.

In order to test this architecture we will time the different connectors under the same conditions. This results will give us the necessary feedback to compare them and conclude which of them are appropriate for which situations.

To summarize, we will start with a theoretical overview of the technologies that will be used in the system, including a brief explanation OpenTSDB, NodeJS, websockets, JavaScript, Python, R, and several visualization frameworks. Then we will explain the architecture of the system as well as some metrics regarding its performance. We will finish with the current state of the system and the future implementations.

1.1 Related work

As we will see in section 3.1, the actual user interface of OpenTSDB is not popular among users. Due to this, there are some other attempts in creating a new web interface. Most of which are local solutions that companies released to the open source community.

There is one solution that stands out from the rest: StatusWolf. It is a front-end made in PHP by the company Box. It has user management and sharing plots and dashboards as main advantages. The main disadvantage of this front-end is its early stage of development that makes it difficult to integrate it with other programming languages to provide the flexibility that the dashboard requires. In Figure 1.1 we can see a screenshot of a form to create a plot in StatusWolf with the main parts highlighted.

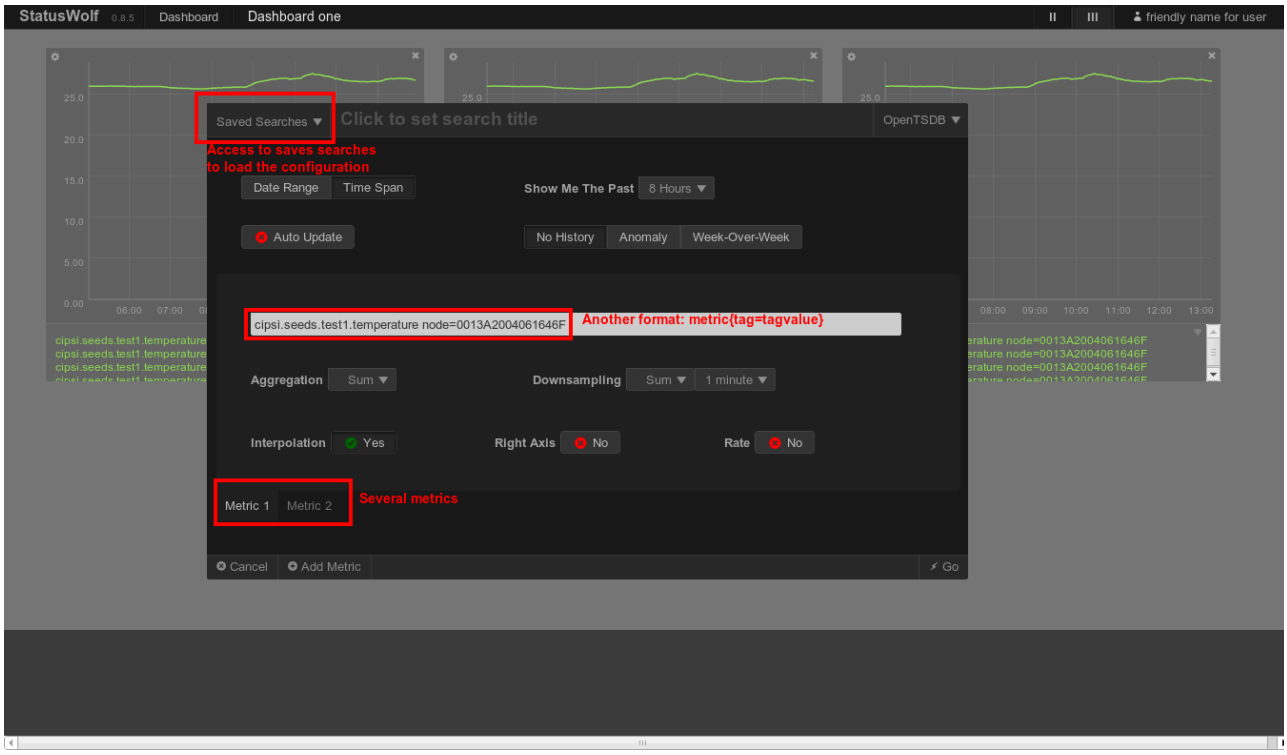


Figure 1.1: StatusWolf interface

Another web visualization interface is Metrilyx, made in Python by the company Ticketmaster. In Figure 1.2 we can see a screenshot of the web interface. Its main advantage is that its built for high availability and distribution of architecture. Yet, similarly to StatusWolf, it only has a method to obtain and manage data. Its development is quite early, it started at the end of February of 2014 with only two users contributing to the code.

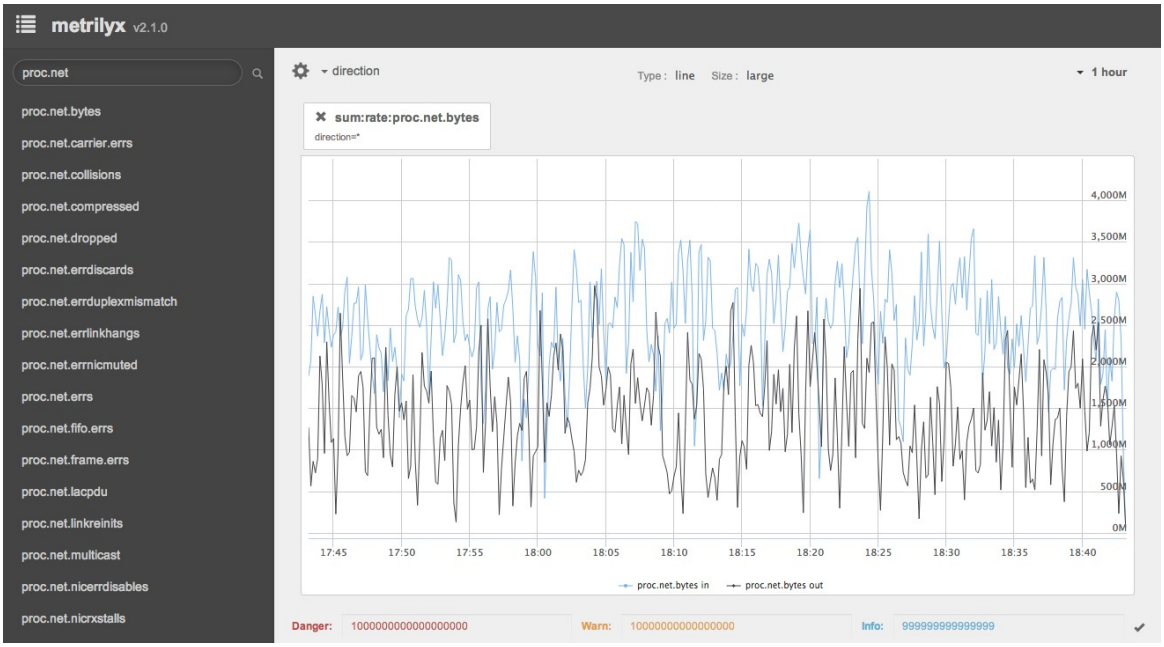


Figure 1.2: Metrilyx web interface

1.2 Organization of the thesis

The organization of the thesis consists of the following:

- Chapter 2 presents a basic background of the technologies that we are going to develop later.
- Chapter 3 shows the solution provided, looking in detail the implementation done.
- Chapter 4 presents how we measured the performance of the proposed architecture
- Chapter 5 has the final conclusions and further work.

2 Theoretical framework

2.1 OpenTSDB

OpenTSDB[1] is a scalable and distributed database built on top of HBase[2]. This database allows us to store high amounts of data at a very high speed, managing insertions every second.

This database provides a user interface that allows the user to browse through all the data. Even though in some aspects it could be useful, it lacks in flexibility and customization.

OpenTSDB provides an HTML API [3] that will allow us to access its data in a standard way.

2.2 Visual analytics

Visual analytics[4] is the science that studies the visual interactive interfaces that facilitate the reasoning of data to an analyst. In visual analytics we find the three-layer architecture[5] (**Figure 2.1**) that a dashboard system should have to fulfill an analyst's requirements in terms of analyzing data. The three layers (analytics, data management and visualization) share information amongst them and integrate in two different workflows: reactive and interactive[6].

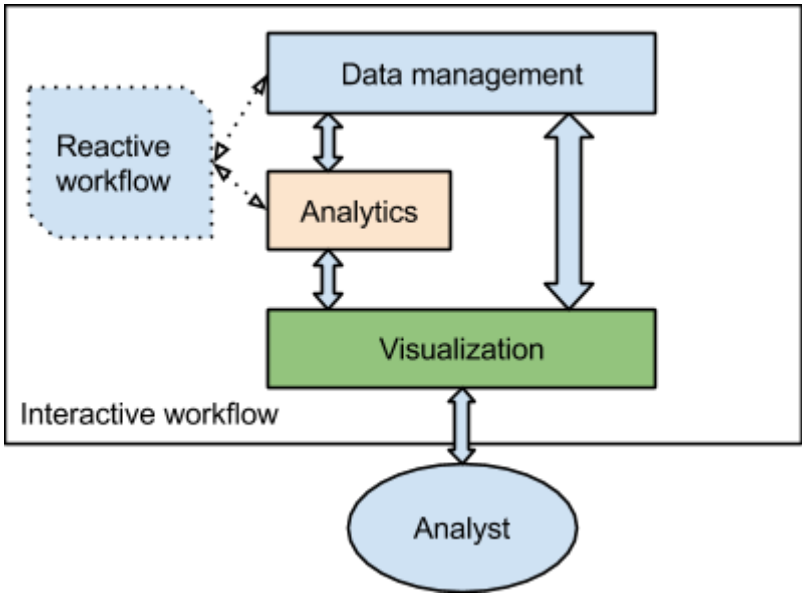


Figure 2.1: Three layer architecture of visual analytics applications with their workflows.

2.2.1 Analytic layer

The analytic layer executes transformations on the raw data such as statistical analysis and predictive behaviour or error detection. It is then able to draw conclusions from it.

2.2.2 Data management layer

The data management layer is responsible for the data lifecycle and procedures of the systems.

2.2.3 Visualization layer

The visualization layer is responsible for taking the requested amount of points and plotting them in an optimal way for the analyst. In web environment, it will refer to the final HTML web page that the analyst will see and interact with. In the web page we will use plotting libraries in javascript that can handle the data points and the interactions of the user.

2.2.4 Workflows: Reactive and Interactive

The reactive workflow runs operations ahead of time to prepare data analysis when the analyst requests it. We can see this workflow in action in real-time applications that need to perform analysis of data as it arrives without interaction from an analyst. For example, in processing images from a camera in real time, the analysis could be performed in the background while the system gets more images.

The interactive workflow run operations when the analyst requests the data. It could be as simple as showing the raw data or as complex as the analyst requests. This workflow requires the interaction of the analyst.

2.3 NodeJS

NodeJS[7] is a web server coded in javascript that allows us to use javascript to create and manage web servers. It is a wrapper around the V8 JavaScript compiler of Google Chrome.

V8 is a highly optimized javascript engine that powers Google Chrome and it is the base of Node, adding Node bindings like sockets and the node standard library that adds more features to V8 to make it a real server solution. The main features that make Node very attractive to deploy web-based applications are asynchronous I/O operations, memory and CPU efficiency, and a strong concurrency handler.

Node is not the first implementation of a server in javascript. However, this implementation has grown exponentially in the last year and has a demonstrated degree of success. The community surrounding Node is growing each day and this popularity is seen in all the modules that are being released into Node, from security modules to real time streaming features. Node has proven high reliability and performance managing web servers. A quick comparison with an Apache web server gives us the conclusion that using Node is faster and more scalable[8]. This technology is being used now by relevant companies such as Google in Google+, Microsoft in Windows Azure interface, or Yahoo in Yahoo Manhattan[9].

2.4 JavaScript

JavaScript is a scripting programming language widely used in the client-side interpreted by web browsers. It appeared in 1995 and was developed by Brendan Eich. JavaScript was developed to interact with the elements of a web page and to be interpreted by the compilers in browsers. It become very popular thanks to the speed and optimization of the web browsers, making their compilers faster and better. These improvements made it suitable for larger and more complex web applications and for the creation of NodeJS. It is a very dynamic programming language with an easy learning curve.

We will use JavaScript as main programming language for the core of the system and for two of the connectors.

2.5 Model View Controller pattern

The Model View Controller (MVC) is a software architecture pattern that focus on modularity and readability. It has three main parts with different functions: the user interacts with the controller, then the controller manipulates the model that updates the view, and finally the user sees it. Is a cycle that repeats with every interaction with the user.

We will implement our solution following this pattern and divide the code accordingly, this will increase the comprehension and flexibility.

2.6 Express

Express[10] is a very popular web framework for nodejs. It is flexible and powerful and allows a user to develop a strong web application spending less time in common problems and patterns.

We used this framework to develop our nodejs server.

2.6.1 hogan-express

Hogan[11] is an HTML template engine that uses Mustache[12]. hogan-express[13] is the module of express for using this template engine in express.

2.7 WebSockets

WebSockets is a technology that allows a full-duplex communication channel using a single socket over the web[14]. With websockets we can create a persistent connection between server and client that allows both to send data. To start the connection, one of the parties must initiate the handshake protocol.

WebSockets are more efficient than HTTP request. Although both HTTP and WebSockets have equivalently sized initial handshakes, websockets only uses that size in the initialization, the rest of the messages have a smaller header[15]. Meanwhile, HTTP messages have large headers throughout.

We will use the Websocket library for NodeJS that allows us to create and manage websockets in a simple and transparent way. We will use the connection to send the points from the server to the client to be drawn.

2.8 Grunt

Grunt[16] is a JavaScript task runner. It automates javascript tasks so the analyst does not have to manually run the requests. We use Grunt in the project to destroy and create a new database, create the entities, and populate with a couple of examples. It is a simple but powerful tool.

2.9 MongoDB

MongoDB[17] is an open-source NoSQL database with document-oriented storage. Its data is encoded in BSON, a binary codification for JSON objects. It is a popular database in the field and has excellent support and documentation to integrate it with NodeJS.

2.10 jQuery

jQuery[18] is a popular library for JavaScript. It simplifies and increases the readability of the javascript in the client-side. We will use from common HTML manipulations to AJAX petitions (asynchronous HTTP petitions) to the API of OpenTSDB.

2.11 Bootstrap

Bootstrap[19] is a responsive and flexible front-end framework created by Twitter that provides interactive and user friendly web interfaces. We will use it as the front-end in our project.

2.12 Highcharts

Highcharts[20] is a JavaScript library used to create interactive charts. It allows you to interact with the data in several ways, like zooming in on a specific part of the chart, adding more contextual information and more features. In our case we will use the time-series plots that Highcharts has to visualize and enhance our data.

2.13 Python

Python[21] is a very popular, high-level programming language developed by the Python Software Foundation in the year 1991. Python is currently utilized in a wide range of applications: being the core of simple scripts, content management systems (CMS) like Drupal, bioinformatics programs, or high-reliability systems.

We decided that Python would be a ideal fit to manage data points from opentsdb because of its demonstrated performance working with data points.

2.14 R

R[22] is a programming language oriented to statistical computing and plotting. It is being used worldwide in statistical analysis because it has many libraries which are useful in that field. We will provide a system that will allow the user to add any function in R to manage and analyse the data points from OpenTSDB.

2.14.1 Opentsdbr

Opentsdbr[23] takes advantage of the HTML API of OpenTSDB to provide a simple, read-only R library to access OpenTSDB. The main disadvantage of this library is that is not optimized.

2.14.2 Rserve

Rserve[24] is a server that provides an API to execute R code from other programming languages. We will use Rserve as middleware between our NodeJS server and R code, allowing us to execute R code to fetch and manage data points from OpenTSDB.

The current version of Rserve (1.7-3) allows to send the data through websockets as well as improved security using HTTPS when sending data.

2.14.3 Node-RIO

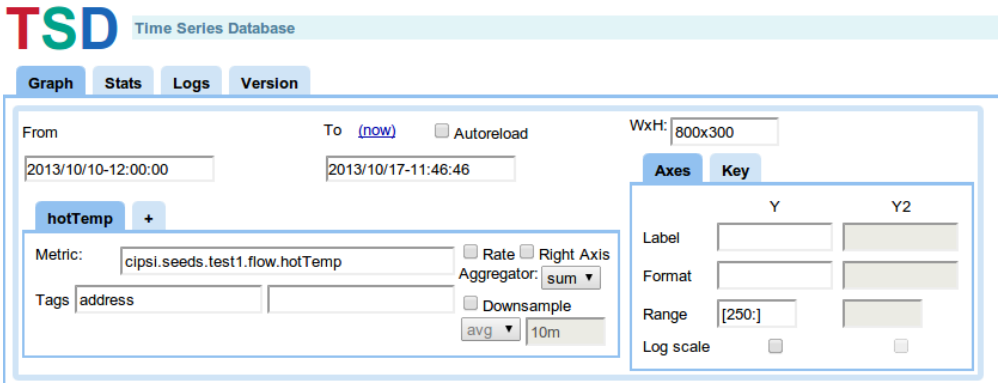
Node-RIO[25] (R In Output) is a NodeJS module that implements the client in JavaScript to interact with Rserve. It abstracts the HTTP request to connect with Rserve.

3 Design and Methodology

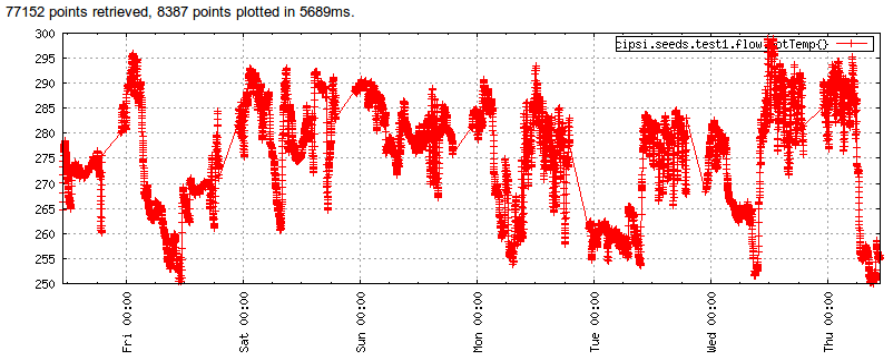
3.1 Actual OpenTSDB interface

The current web interface of OpenTSDB consists of two main parts **(Figure 3.1)**: the form to enter the query **(Figure 3.1A)** and the resulting plot **(Figure 3.1B)**. When we access the interface, the blank form shown in **(Figure 3.1A)** appears and it is ready to accept parameters. Once the parameters are filled in, a plot, such as the one seen in **Figure 3.1B**, appears. This plot presents a huge disadvantage to OpenTSDB users because it is only an image. If there was a mistake in entering the scale **(Figure 3.1C)** or size, the query would have to be executed again.

A



B



C

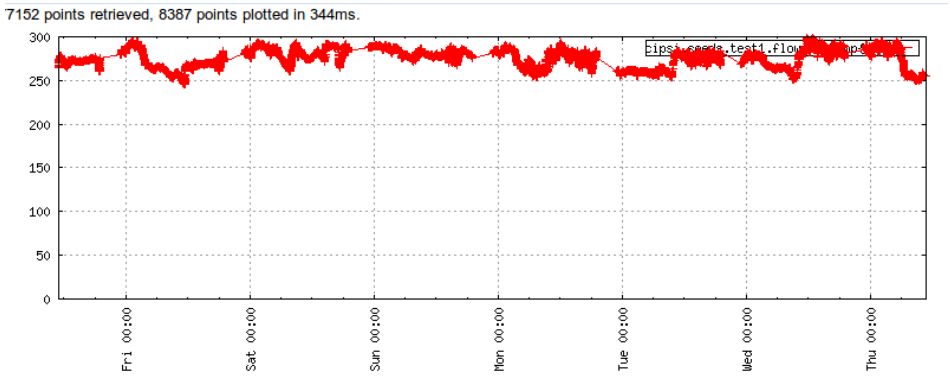


Figure 3.1 shows the current UI of the OpenTSDB system. **A)** shows the query window, **B)** the resulting plot, and **C)** the same plot in wrong scale (starting in 0)

In order to measure the general satisfaction and usage of a common user of OpenTSDB, we created a poll to ask the community about it. We designed an online questionnaire and shared it through the main list. We gathered some interesting results:

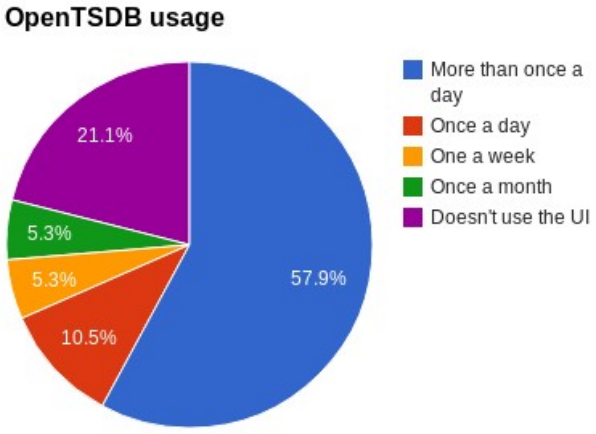


Figure 3.2: OpenTSDB usage poll

When asking about usage **(Figure 3.2)** we see that more than half (57%) use it more than once everyday and when asking about their satisfaction we found that 36.8% rate the interface 2 out of 5.

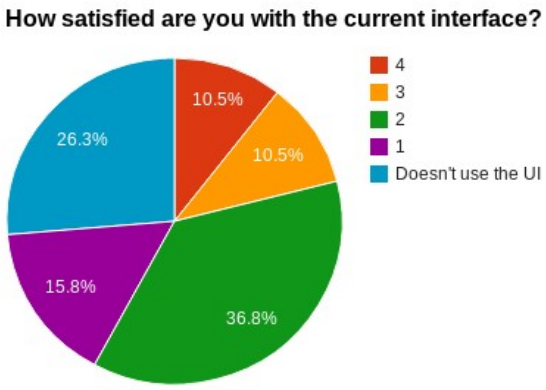


Figure 3.3: OpenTSDB satisfaction poll

As we see in **Figure 3.3**, there is a general dissatisfaction with the current user interface, even when the most of the users use it more than once a day.

3.2 Main server structure

For the proposed system, we created openTSDBnode, an entire NodeJS server, with user authentication, following the Model View Controller (MVC) pattern. We used bootstrap for making a user friendly interface. For the connection with different languages we created different libraries, like nodetsdb (nodejs - opentsdb), nodetsdb-client (nodejs - client - opentsdb), a nodejs wrapper for a python script and another wrapper for the R language.

We decided to follow the model view controller (MVC) pattern in order to make a more readable and maintainable code. In the **Figure 3.4** we can distinguish every component of the MVC pattern.

- app.js is the main file that creates the server and handles all the petitions of the server. In the MVC pattern, app.js is the Controller.
- The routes folder contains all the middleware to handle a specific request, for example in the stats.js, it will take care of query opentsdb about its stats, organize them and then call the appropriate view to show it. The route files are the Model in the MVC pattern.
- Finally the easiest to see, the views folder is the View in the MVC pattern. It contains Moustache HTML templates (that are handled by the hogan-express module) that will compose the final HTML page with the information provided by the routes.

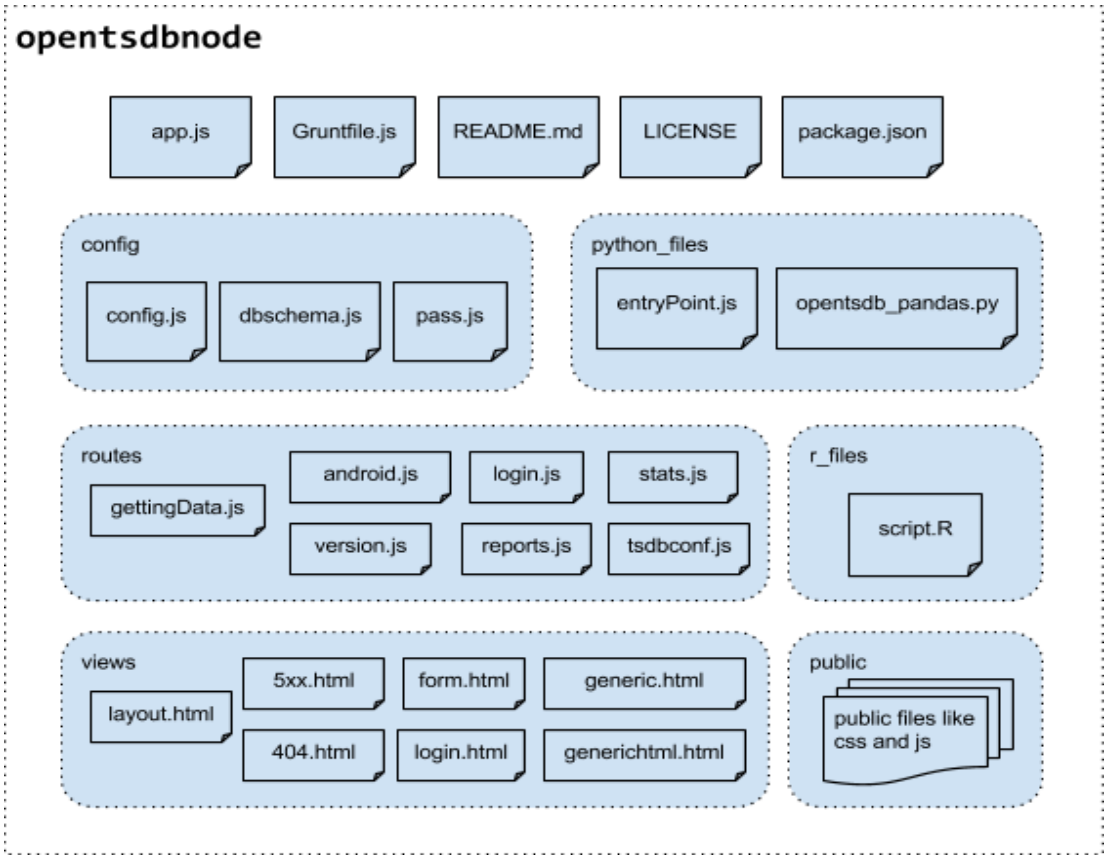


Figure 3.4: Main file structure of opentsdbnode

3.2.1 app.js

This is the main file of the server where we load all the node modules, make the proper configuration and start the server in the correct port. In the first part of the file we declare all modules that we are going to use later. Here we show some of the most important modules in the file.

```
/**
 * Module dependencies.
 */
var express = require('express'), //NodeJS framework, the first one
 app = express(), //Initialize the express module
 db = require('./config/dbschema'), //MongoDB schemas
 pass = require('./config/pass'), //User authentication configuration
 passport = require('passport'), //User authentication module
 config = require("./config/config"), //Main configuration of opentsdbnode
 login = require('./routes/login'), //Login model
 stats = require('./routes/stats'), //OpenTSDB stats model
 testData = require('./routes/gettingData'), //Model to obtain data from tsdb
 android = require('./routes/android'), //Model for the android API
 reports = require('./routes/reports'), //Model for handeling the reports
 http = require('http'), //Module for http requests
 nodetsdblib = require('nodetsdb'), //Module for connecting node-opentsdb
 path = require('path'), //Module to manage paths in the system
 io = require('socket.io'); //Module to use WebSockets
```

The next section of the file is dedicated to the configuration of the express framework:

```
/**
 * Setting environments for express.
 */
app.set('port', process.env.PORT || 3000); //Set the port
app.set('views', path.join(__dirname, 'views')); //Set where are the views
app.set('view engine', 'html'); //Setting HTML as filetype view
app.set('layout', 'layout'); //Setting the file "layout" in views as the layout
app.enable('view cache'); //Enabling cache in express
app.engine('html', require('hogan-express')); //hogan-express as HTML engine
app.use(express.favicon());
app.use(express.cookieParser()); //Enabling cookies
app.use(express.session({ secret: 'sweetieKittyCat' })); //Session secret
app.use(passport.initialize()); //Enabling account management security
app.use(passport.session());
app.use(passport.authenticate('remember-me'));
```

Once we have all the environments initialized correctly we can start to configure the behaviour with different endpoints. This is one advantage of express as a framework: adding endpoints is a matter of adding one more line of code. For example:

```
app.get('/login',login.sign);
```

That means that every time there is a request for the page www.ourserver.com/**login**, the controller (app.js) will receive this request and pass it to the function "sign" of the login module (declared where the modules were declared). Another important security feature is that this line also implies that there is no need of authentication from the user to access that specific view. If we wanted to secure a view to be only accessible for authenticated users, we just need to add another variable to the function:

```
app.get('/statistics',pass.ensureAuthenticated,stats.stats);
```

Adding "pass.ensureAuthenticated" we ensure that the page www.ourserver.com/**statistics** is only going to be accessible for authenticated users. Those are not the only ways to create an endpoint. If it is not needed to invoke a model, it is possible to handle the request directly in the controller.

```
//End point to remove a report of the database
app.get('/removereport', function(request, res){
   //Retrieve the parameter id from the GET request
   var id = request.query.id;
   if(id){
         /**
          * If the id was in the query, we delete it
          * db is the object representing the db
          * reportModel is the model of a report in the db
          */
      db.reportModel.remove({ _id: id }, function (err) {
       if (err){
        /**
            * If there was an error deleting
            * the report we render the generic
            * view with the error.
            */
         res.locals.title = 'Remove report';
         res.locals.block= 'Error deleting the report';
         res.render('generic');
       }else{
        /**
            * If there were not any errors
            * we render the view directly
            * with the correct parameters
```

```
            */
          res.locals.title = 'Remove report';
          res.locals.block= 'Report deleted correctly';
          res.render('generic');
        }
       });
    }else{
           /**
            * If the id is not in the request
            * we render the view with
            * the error
            */
         res.locals.title = 'Remove report';
         res.locals.block= 'Error deleting the report, no id provided';
         res.render('generic');
        }
});
```

The previous function will receive a GET request like www.ourserver.com/removereport?id=11 and will handle everything in the controller, rendering the correct view.

“res.locals.title = 'Remove report';” and “res.locals.block= 'Error deleting the report, no id provided';” are filling two variables of the view “generic” that all together will merge in the template declared before in app.set('layout', 'layout');.

Nearly at the end of the file app.js we found the line of code that initializes the server.

```
var server = http.createServer(app).listen(app.get('port'), function(){
  console.log('Express server listening on port ' + app.get('port'));
});
```

We create the server listing in the port we specified before in the configuration area and we log it in the terminal.

Lastly, we have to initiate and configure the websockets. To initiate them we simply call:

```
var websocket = io.listen(server,{ log: false });
```

This will make the websocket module listen to any petition related to websockets that comes to this server. Then we just need to configure them to handle those requests. Here we have one example of the websockets configuration.

```
websocket.sockets.on('connection', function (socket) {

  socket.on('getDataPoints', function (options) {
    var data = {
          metric:test1.temperature',
          start: {timestamp:'2013-08-04 12:00:00', timezone:'CEST'},
          end: {timestamp:'2013-08-07 14:00:00', timezone:'CEST'},
          tags:[{name:'node', value:'0013A2004061646F'}],
          debug:true
       }

    executeRio(data,function(result){
       var time2 =  new Date().getTime();
       socket.emit("dataServer",result); });
   });
});
```

Initially, we start the websockets when we have a connection, then if the request is "getDataPoints" (in this particular case) we will execute some code that will (but not necessarily) end emitting through the websockets with the code "dataServer" the result, socket.emit("dataServer",result); });. The other party (the client in this case) will have similar code to emit and handle the request.

3.2.2 Routes

The files in routes are the middleware that will handle a specific request sent by the controller (app.js). In **Figure 3.5** we can see the main routes.

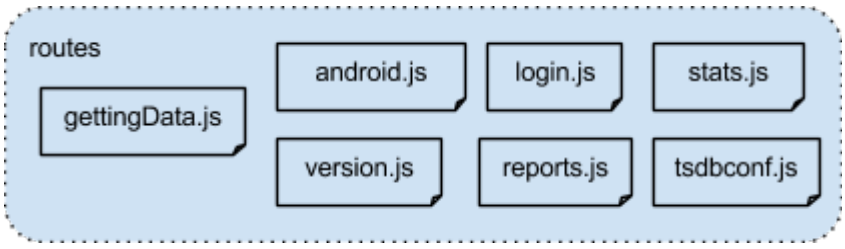


Figure 3.5: Main files in routes

To explain the main structure of a route, we will take the version.js file as example.

```
/*
 * GET tsdb version page.
```

```
 */
var config = require("../config/config");
/*The exports object allows us to add a function
*to itself so later we can call it with require like:
*        var tsdbversion = require("./routes/version.js")
*        tsdbversion.version(req, res);
*/
exports.version = function(req, res){
  var blocks ="";
  //We do a GET request to opentsdb to get the opentsdb version

http.get("http://"+config.opentsdbserver+":"+config.opentsdbserverport+"/version",
function(ress) {
     console.log("Got response: " + ress.statusCode);
     ress.on('data', function (chunk) {
        //We obtain the answer of the server and then we render the generic view with
the data
        res.render('generic', { title: 'OpenTSDB version', block:chunk,user: req.user });
     });

  }).on('error', function(e) {
     //If there was an error in the connection, we render the generic view with the
error
     console.log("Got error: " + e.message);
     res.render('generic', { title: 'OpenTSDB version', block:"Connection error",user:
req.user });
     });
};
```

In the routes the most important thing is to associate the object you want to use from the controller with the object “exports”. We have to associate an object (or function) as a property of this object, so later we can call it with a require statement. In this case we associate the function that will receive the request (req) object and the response one (res) and that will handle the request. Then in the controller (app.js) we can refer to it in the modules declaration area:

```
var versionobj = require('./routes/version'), //OpenTSDB version model
```

And call it later to handle the request of www.ourserver.com/tsdbversion

```
app.get('/tsdbversion',pass.ensureAuthenticated,versionobj.version);
```

We call the function version from the object versionobj. In this call it does not need any parameter because the app.get function will add the request and response.

A route file could be even more simple, for example the route login.js

```
/*
 * GET login page.
 */

exports.sign = function(req, res){
  res.render('login',{layout: '', user: req.user, message:req.flash('error') });
};
```

In this case the route only has to render the login view passing the user id (for authentication purposes) and a message that could be an error message if a user tried to authenticate and the password was wrong.

However, a route does not always have to render a view. The route android.js is a simple API created to help a bachelor student at the University of Stavanger. The details of implementation are available at the section 3.2.7.

3.2.3 Views

The views in opentsdbnode are HTML files with Mustache notations that are going to be handled by hogan-express, a node module that interprets and manages HTML files with mustache notations. In **Figure 3.6** we can see the main views of opentsdbnode.

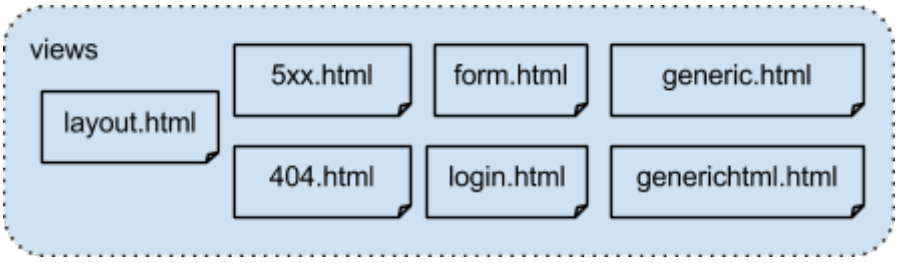


Figure 3.6: Main views of opentsdbnode

To use the views, first we have to declare it in the app.js file

```
app.set('views', path.join(__dirname, 'views')); //Set where are the views
app.set('view engine', 'html'); //Setting HTML as filetype view
```

```
app.set('layout', 'layout'); //Setting the file "layout" in views as the layout
app.engine('html', require('hogan-express')); //hogan-express as HTML engine
```

Those are the main configuration lines in order to use HTML files handled by the hogan-express module. A special mention to the way of setting the template layout, the file “layout”. It contains the header and the footer of the output in HTML.
The first part of the file is the “head” with all the CSS declaration.

```
<head>
    <title>openTSDBnode</title>
    <!-- Bootstrap -->
    <link href="stylesheets/bootstrap.min.css" rel="stylesheet" media="screen">
    <link href="stylesheets/bootstrap-responsive.min.css" rel="stylesheet"
media="screen">
    <link href="stylesheets/styles.css" rel="stylesheet" media="screen">
    <link href="stylesheets/datepicker.css" rel="stylesheet" media="screen">

</head>
```

And in another part of the code, closer to the footer we can find the main Mustache notation.

```
<div class="container-fluid">
        <div class="row-fluid">
          <div class="span9" id="content">
            <div class="row-fluid">
                    {{{ yield }}}
            </div>
          </div>
        </div>
        <hr>
        <footer>
          <p>Roberto Martin</p>
        </footer>
    </div>
```

Later we declare the javascripts that we will need in the client such as jQuery, custom ones like nodetsdbclient.js (see section 3.4 for complete explanation) or plots.js.

```
    <script src="javascripts/jquery-1.9.1.js"></script>
    <script src="javascripts/ui/jquery-ui.js"></script>
    <script src="javascripts/jquery.flip.js"></script>
    <script src="javascripts/bootstrap.min.js"></script>
    <script src="javascripts/bootstrap-datepicker.js"></script>
```

```
    <script src="javascripts/highcharts.js"></script>
    <script src="/socket.io/socket.io.js"></script>
<script src="javascripts/scripts.js"></script>
    <script src="javascripts/nodetsdbclient.js"></script>
<script src="javascripts/plots.js"></script>
```

The {{{ yield }}} tag will be replaced with the content of other views, creating the final HTML output that will be sent to the client. In **Figure 3.7** we can see the main workflow concerning the creation of final HTML that will be sent to the user.

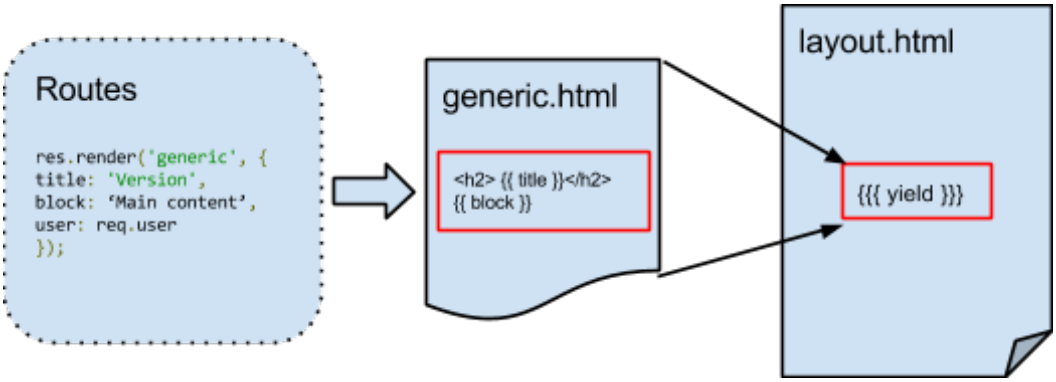


Figure 3.7: General workflow of creating a final HTML

In views there are two files that are very similar, generic.html and generichtml.html

```
<!-- generic.html -->

<h2> {{ title }}</h2>
{{ block }}
```

```
<!-- generichtml.html -->

<h2> {{ title }}</h2>
{{{ block }}}
```

In Mustache, if we use two brackets, it will put that exact string in that place. Even if there is HTML code, it will appear as a string. If we instead use three brackets all the HTML code will be interpreted.

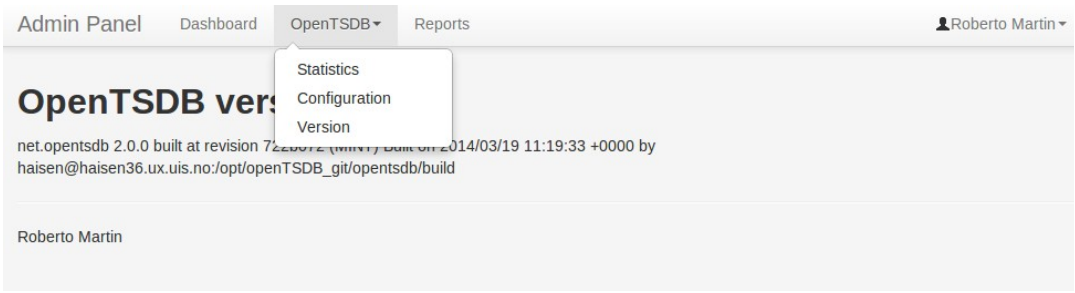


Figure 3.8: OpenTSDB version page

3.2.3.1 Plotting view

One of the most important views is the one that contains the form to obtain and plot data points. It is temporarily is under /form but in future versions it is planned to be moved to a more generic url.

When receiving the /form request, the controller (app.js) renders directly the view form.html

```
app.get('/form', function(req, res){
  res.render('form');
});
```

That will render the HTML shown in **Figure 3.9**

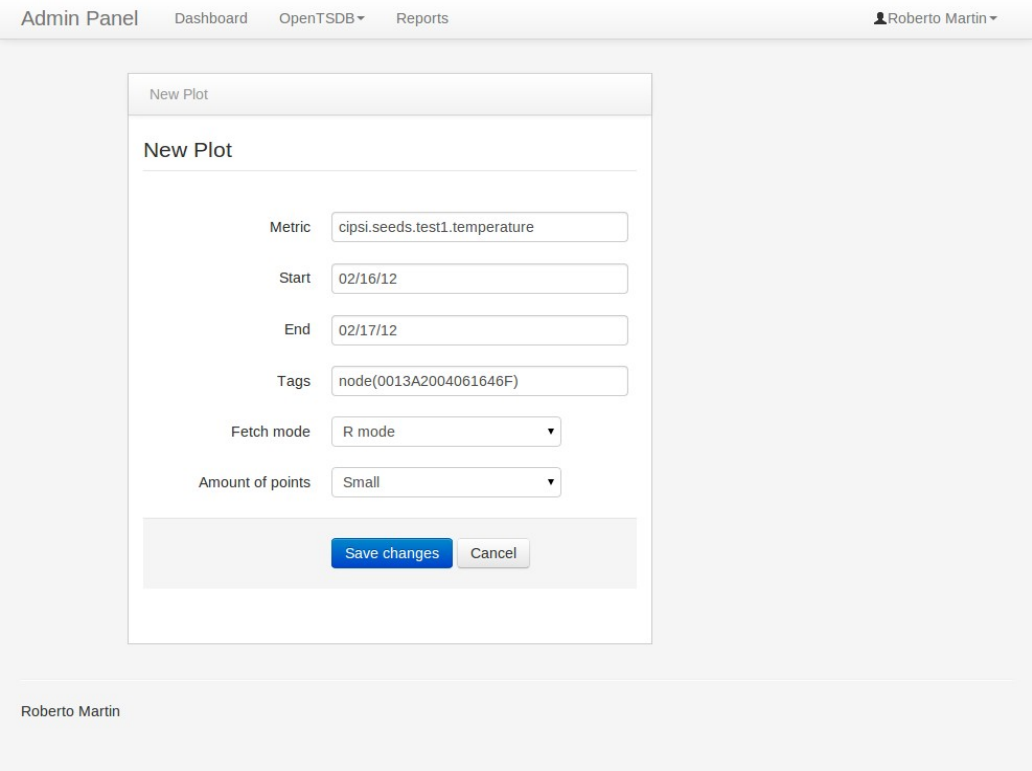


Figure 3.9: Form produced by form.html

The main logic of this form is handled by the javascript file plots.js. When the form is complete and the button is clicked, the function handleClick is the one in charge of managing what kind of connector is the one that is going to take care of the request: through the server, directly from the client, through python, or through R.

```
function handleClick() {
   //Hide the form
   var el = document.getElementById('flipbox');
   el.style.display = 'none';

   var e = document.getElementById("selectMode");
   var strUser = e.options[e.selectedIndex].value;

   var e = document.getElementById("selectAmount");
   var amountPoints = e.options[e.selectedIndex].value;

   switch (strUser){
    case '1':
```

```
            //R mode
            console.log('[handleClick] R mode');
            x="Today is Monday";
            break;
          case '2':
            //Server HTML API mode
            console.log('[handleClick] Server mode');
            handleServer(amountPoints);
            break;
          case '3':
            //Client HTML API mode
            console.log('[handleClick] Client mode');
              handleClient(amountPoints);
            break;
          case '4':
            //Python mode
            console.log('[handleClick] Python mode');
              handlePython(amountPoints);
            break;
          default:
            console.log('[handleClick] Default!');
            break;
          }
          event.preventDefault();
      }
```

The different methods of connection will be explained in more detail in the section 3.3.

In order to test and compare the connectors we created a fixed amount of data points to be fetched (method explained later in the Results section), but that will be discarded in the next version of opentsdbnode.

Once the data points are collected, three of the four methods retrieve them through websockets. The fourth one is the connection with the client, so the data points are already in the client javascript after querying OpenTSDB directly. Then the data points are processed and plotted with the plotting library Highcharts. Using this library enables us to zoom in the data as well as adding extra information on each data point.

On the subject of plotting libraries, they are numerous and we evaluated some according to the requirements of the project:

	Interactiveness	**Mobile friendly**	**Popularity**	**Community**	**Time series specialization**
gRaphaël [26]	Medium	--	Medium	Medium	Low
JavaScript InfoVis Toolkit [27]	Medium	Medium	Low	Low	Low
milkchart[28]	Low	Medium	Low	Low	Low
jQuery Visualize Plugin [29]	High	High	Low	Low	Low
moochart [30]	Low	Medium	Low	Low	Low
JS Charts [31]	Low	Medium	Low	Low	Low
Timeline [32]	Medium	High	Low	Medium	Medium
D3js [33]	High	High	High	High	High
Highcharts	High	High	High	High	High

Table 3.1: Comparison of visualization libraries

After reviewing the libraries in **Table 3.1** , we see that only two meet the requirements, D3js and Highcharts. We will use Highcharts as a visualization engine for testing the architecture. In **Figure 3.10** we can see an example of a zoomable time series plot using Highcharts.

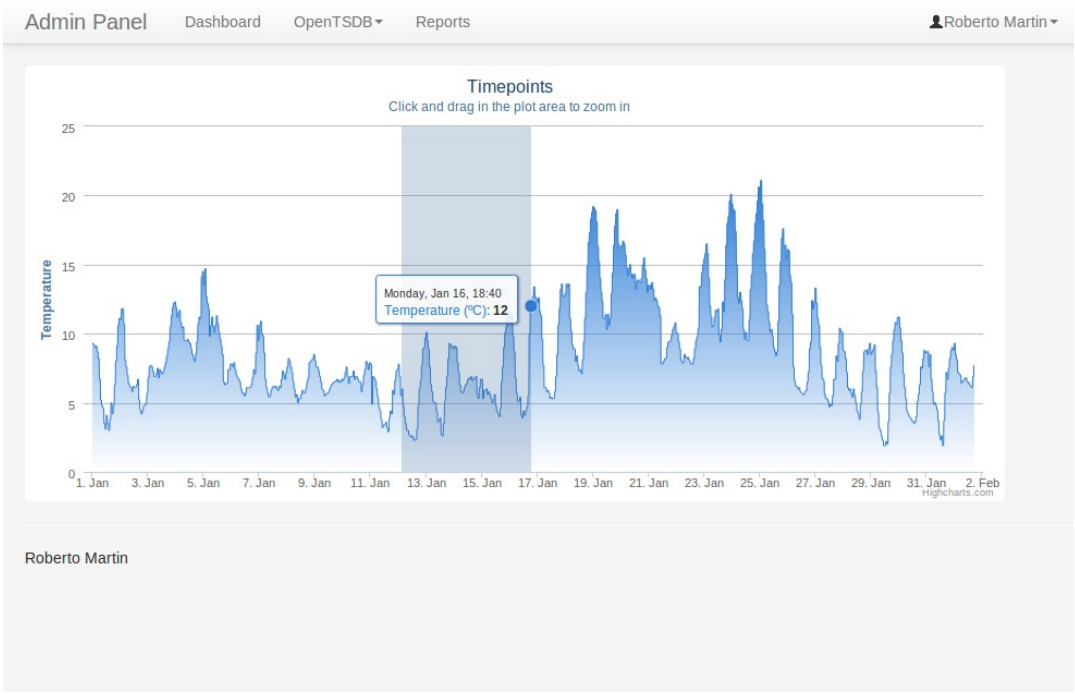


Figure 3.10: Example of plotting data points

3.2.4 Database

For the database we will use MongoDB because of its great integration with node. We will also use the node module “mongoose” that will allow us to interact with the database in a dynamic way. All the configurations of the database are in ./config/dbschema.js

The first thing to do is to connect to the database and then create the schemas needed. A schema is an abstract representation of the object that we want to store. It can have specific methods, preconditions and postconditions when inserting in the DB, in addition to more features. In our case we will create a user schema (for account management), token schema (login management), and report schema (help with timing the tests).

```
// Database connect
var uristring =
```

```
  process.env.MONGOLAB_URI ||
  process.env.MONGOHQ_URL ||
  'mongodb://localhost/test';

var mongoOptions = { db: { safe: true }};

mongoose.connect(uristring, mongoOptions, function (err, res) {
  if (err) {
    console.log ('ERROR connecting to: ' + uristring + '. ' + err);
  } else {
    console.log ('Successfully connected to: ' + uristring);
  }
});
```

Before starting to create the specific schemas, we have to obtain the global object "Schema".

```
//Database schema
var Schema = mongoose.Schema,
        ObjectId = Schema.ObjectId;
```

3.2.4.1 User schema

The user schema is created to represent an account of the system. It will be used for authentication and for personalised configurations.

In order to create the schema, we have to declare it as follows.

```
// User schema
var userSchema = new Schema({
  username: { type: String, required: true, unique: true },
  email: { type: String, required: true, unique: true },
  password: { type: String, required: true},
  admin: { type: Boolean, required: true },
});
```

Our user object will have a username, email, password, and a boolean to see if it is an admin. It is a simple schema that will be improved in following versions.

Methods for authentication are also required: one to hash the password before storing, another method to compare, and another to generate a random token that will be used to maintain the session in the server. We will use the node module Bcrypt that contains the cryptographic functions needed.

```
// Middleware for password
userSchema.pre('save', function(next) {
        //Before saving the user we hash the passsword and save it
        var user = this;
        if(!user.isModified('password')) return next();
        bcrypt.genSalt(SALT_WORK_FACTOR, function(err, salt) {
                if(err) return next(err);
                bcrypt.hash(user.password, salt, function(err, hash) {
                        if(err) return next(err);
                        user.password = hash;
                        next();
                });
        });
});

// Password verification
userSchema.methods.comparePassword = function(candidatePassword, cb) {
        bcrypt.compare(candidatePassword, this.password, function(err, isMatch) {
                if(err) return cb(err);
                cb(null, isMatch);
        });
};

// Session management implementation helper method
userSchema.methods.generateRandomToken = function () {
  var user = this,
                                                        chars          =          "_!
abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ1234567890",
      token = new Date().getTime() + '_';
  for ( var x = 0; x < 16; x++ ) {
    var i = Math.floor( Math.random() * 62 );
    token += chars.charAt( i );
  }
  return token;
};
```

After defining the user schema, we need to export it so it will be available in other scripts to add, find, or delete a user.

```
// Export user model
var      userModel      =      mongoose.model('User',
userSchema);
exports.userModel = userModel;
```

3.2.4.2 Token schema

The token schema is just to associate the user to a unique token that will be used to maintain the session. The token will be shared with the client and will allow to keep the login for an amount of time.

```
var tokenSchema = new Schema({
  accessToken: { type: String, required: true, unique: true },
  usernameid: { type: String, required: true, unique: true },
});
```

This schema only needs an extra function that will allow to consume the token once the expiration time is up. And as with the user schema, we have to export it.

```
tokenSchema.methods.consumeRememberMeToken= function(token, fn) {
  var uid = token.usernameid;
  token.remove();
  return fn(null, uid);
};

var tokenModel = mongoose.model('Token', tokenSchema);
exports.tokenModel = tokenModel;
```

3.2.4.3 Report schema

In order to obtain timing results faster and in an efficient way, we created a report system explained in detail in the section 3.2.5. For this reporting system, we needed to create a report schema in order to store the timing of different connectors of opentsdbnode.

This is a simple schema, that does not need any extra functions.

```
//  ========== Report Schema ==========

var reportSchema = new Schema({
  method: { type: Number, required: true}, // 1:R, 2:Server, 3:Client, 4:Python
  testgroup: { type: Number, required: true},
  dpsize: { type: Number, required: true},
  stage: { type: Number, required: true},
  description: { type: String, required: true},
  time: { type: Number, required: true}, //ms
});

var reportModel = mongoose.model('Report', reportSchema);
exports.reportModel = reportModel;
```

3.2.4.4 Grunt

During the development of opentsdbnode we needed to drop the database and recreate the schemas with some users as tests. For these tasks we used Grunt.

In order to use it we just need to add a file named Gruntfile.js in the root of the project and register the tasks that should be done.

```
var db = require('./config/dbschema');
module.exports = function(grunt) {
    grunt.registerTask('dbdrop', 'drop the database', function() {
        // async mode
        var done = this.async();
        db.mongoose.connection.on('open', function () {
          db.mongoose.connection.db.dropDatabase(function(err) {
            if(err) {
              console.log('Error: ' + err);
              done(false);
            } else {
              console.log('Successfully dropped db');
              done();
            }
          });
        });
      });
};
```

With this code, we just need to execute in the terminal

```
grunt dbdrop
```

3.2.5 Reporting system

In order to centralize all the necessary timing to be able to compare the different kinds of connections, we created a system to store it. For that, we created three endpoints in the server: one to add a new time, another one to delete a specific one, and another one for listing the existing timings. This will allow us to send requests to the server from the client or from our own server and have them all listed in a specific page.

To save a report we need to send a POST request with all the parameters to the endpoint "/saveReport" of opentsdbnode. The controller (app.js) will manage the request and if all the parameters are there[1], it will call the following function to save the new report in the database.

```
function saveReport (meth, tg, dpsz, stg, desc, t){
    var report = new db.reportModel({ method: meth
                                    , testgroup: tg
                                    , dpsize: dpsz
```

1 If all of the parameters are not there, an error page is rendered.

```
                                    , stage: stg
                                    , description: desc
                                    , time: t});

    report.save(function(err) {
      if(err) {
        return 0;
      } else {
        return 1;
      }
    });
};
```

If we need to remove a report, we have to send a GET request to the "/removereport" endpoint.

```
//End point to remove a report of the database
app.get('/removereport', function(request, res){
    //Retrieve the parameter id from the GET request
    var id = request.query.id;
    if(id){
          /**
           * If the id was in the query, we delete it
           * db is the object representing the db
           * reportModel is the model of a report in the db
           */
        db.reportModel.remove({ _id: id }, function (err) {
          if (err){
            /**
                * If there was an error deleting
                * the report we render the generic
                * view with the error.
                */
             res.locals.title = 'Remove report';
             res.locals.block= 'Error deleting the report';
             res.render('generic');
          }else{
            /**
                * If there were not any errors
                * we render the view directly
                * with the correct parameters
                */
            res.locals.title = 'Remove report';
            res.locals.block= 'Report deleted correctly';
            res.render('generic');
          }
        });
    }else{
           /**
            * If the id is not in the request
            * we render the view with
```

```
            * the error
            */
          res.locals.title = 'Remove report';
          res.locals.block= 'Error deleting the report, no id provided';
          res.render('generic');
         }
});
```

And finally to list all the reports, we have to access to "/reports" and if there are reports saved in the database, it will list them as shown in **Figure 3.11**.

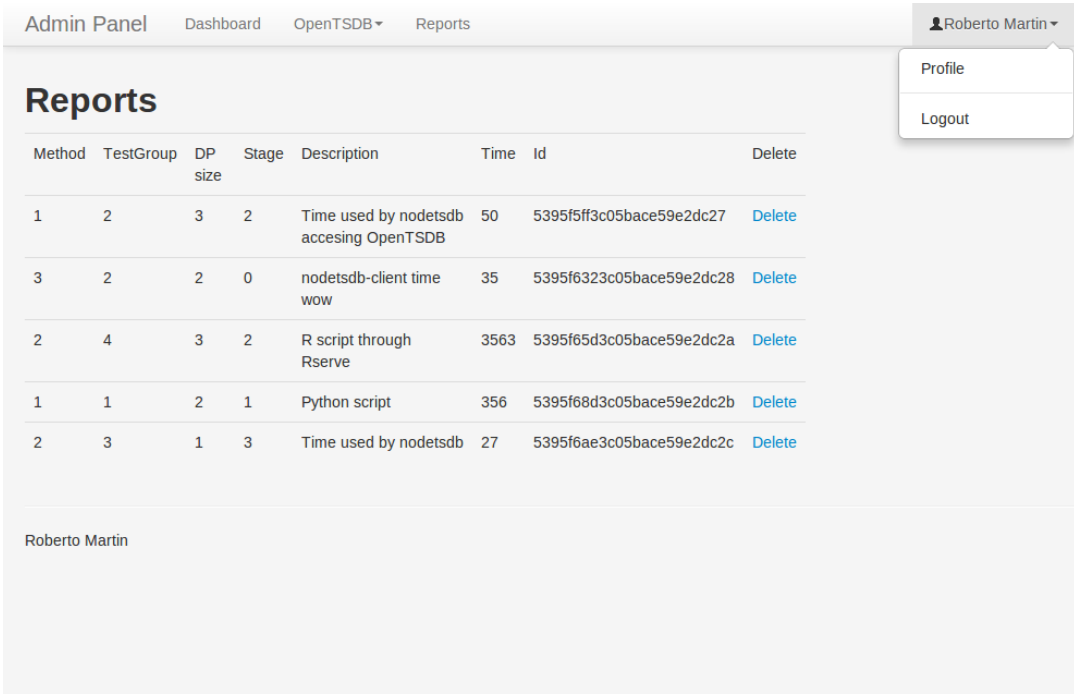


Figure 3.11: Example of a list of reports.

3.2.7 Other features

3.2.7.1 Android endpoint

opentsdbnode also contains an endpoint needed by a bachelor student of the University of Stavanger in order to finish his bachelor thesis. For implementing the endpoint, we create a new route android.js.

```
/*
 * Android json model.
 */
var config = require("../config/config");
var nodetsdblib = require('nodetsdb');

exports.getData = function(req, res){
  console.log('Preparing query');
  var blocks ="",
   start = req.query.start,
   end = req.query.end,
   metric = req.query.metric,
   aggregator = req.query.aggregator;

  var nodetsdb = new nodetsdblib({host:config.opentsdbserver,
port:config.opentsdbserverport});

  var queryconf = {start:start,end:end, metric:metric, aggregator:aggregator,
tags:{}};

    nodetsdb.getDataPoints(queryconf, function(dp){
      if(dp){
         res.contentType('application/json');
         res.send(dp);
     }else{
         res.contentType('application/json');
         res.send({error:'Error or empty'});
     }
  });
};
```

In this case the route receives the GET request and extracts the parameters from the request object:

```
start = req.query.start,
```

Then we use the module nodetsdb (explained in detail in the chapter 3.3) to get data points from OpenTSDB and if everything is without errors, instead of rendering a view, we render a JSON file, so the client (the android phone in our case) could parse it and extract the data points. It was decided to put opentsdbnode as middleware between OpenTSDB and the android clients in order not to expose OpenTSDB on the web. With this configuration, opentsdbnode will act as the gateway between them.

3.2.7.2 Configuration file

In order to modularize the code we created a configuration file in ./config/config.js that contains the OpenTSDB address and port. If it is needed to change them it is only necessary to make changes there. To access it, we first declare a module

```
var config = require("../config/config")
```

and then simply access its properties:

```
[...] config.opentsdbserver+":"+config.opentsdbserverport [...]
```

It is a simple file, that can be expanded easily:

```
var config = {}

config.opentsdbserver="haisen36.ux.uis.no";
config.opentsdbserverold="haisen23.ux.uis.no";
config.opentsdbserverport="4242";

module.exports = config;
```

3.2.8 License

We decided to use a open source license the BSD 3-Clause License, to allow others to redistribute, improve, or use the code without problems. In continuation, we can see the full license explained:

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

* Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.

* Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.

3.3 Connectors

The most important parts of opentsdbnode are the connectors. opentsdbnode has four different ways of connecting and obtaining data points from OpenTSDB, each of them with their advantages and disadvantages that we will see in following sections.

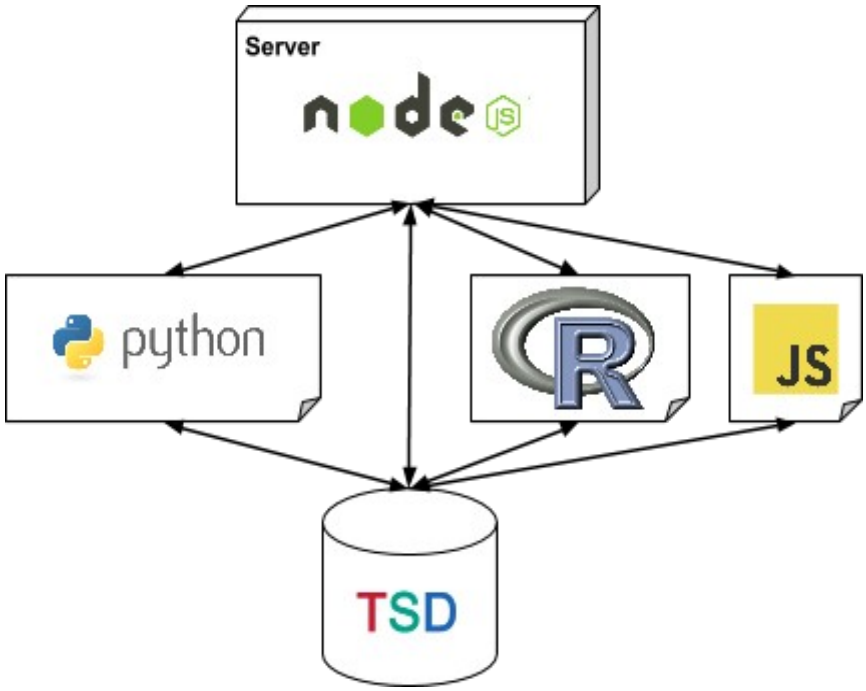


Figure 3.12: General overview of the connectors of opentsdbnode

We have implemented each connector having in mind visual analytics. **Figure 3.12** shows the general architecture of the system proposed. During the following sections, we will explain each connector in detail as well as how its pieces match with the components of visual analytics.

3.3.1 NodeJS - OpenTSDB connector (nodetsdb)

This connector is the link between the node server and OpenTSDB; the client has to send a request through websockets and then the server will fetch the data points with the configuration that the client sends using the library nodetsdb, built specifically for opentsdbnode.

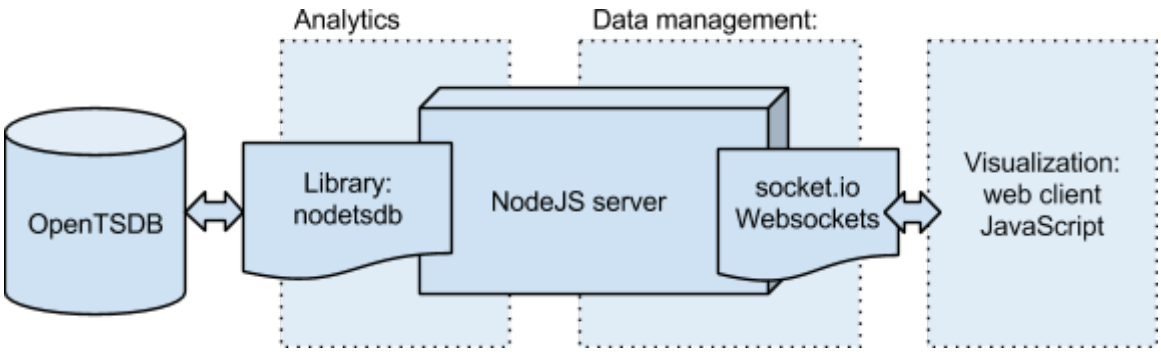


Figure 3.13: Structure of the nodetsdb connector, showing the visual analytics sections

In **Figure 3.13** we can see how the sections of the code match with the theoretical visual analytics parts. The analytics section will be the library nodetsdb used to fetch the data and the part of the node server that executes it. Once we fetch the data points, we have the data management section. This part could be improved later adding more data managing in JavaScript, like filters or data transformation functions.

The following code in app.js is responsible for use of the module nodetsdb and emittance of results through the same socket that the client used to request them. The client is in charge of closing the socket when it receives them.

```
/* ServerMode */
    socket.on('getDPServerMode', function (options) {
       var nodetsdb = new nodetsdblib({host:config.opentsdbserver,
                                port:config.opentsdbserverport });
       nodetsdb.getDataPoints(options, function(dp){
          if(dp){
            //There are datapoints
            socket.emit("dataServer",dp);
          }else{
            //There are not datapoints
            console.log('Sorry no datapoints');
          }
       });
    });
```

This connector has several advantages.

- The client does not need to be connected with OpenTSDB directly.
- The client is using opentsdbnode as proxy between them. It allows a better control in the access of OpenTSDB.

- It puts the heavy work on the server, making a lightweight client only responsible for plotting the results.
- The possibility of caching data points, detecting patterns, or the ability to save favourites queries.

3.3.1.1 nodetsdb

nodetsdb is the library created to wrap all the connection with OpenTSDB. It is hosted in github[34] with a BSD Clause-3 license like opentsdbnode. It is linked with npm[35] , the repository of node modules, so anyone can add it to their node projects with a simple command.

```
npm install nodetsdb --save
```

It consists of a javascript object that contains two properties (OpenTSDB host and port) and a method to get data points from a specific query.

```
var Nodetsdb = function(configuration){
   if(!configuration.host || !configuration.port){
      throw 'Please provide a host and a port';
   }

   this.host = configuration.host;
   this.port = configuration.port;

   this.getDataPoints = function(query, callback){
[...]
```

The function to obtain the data points first creates the correct format of the query and then does an HTTP request to the OpenTSDB server.

```
 this.getDataPoints = function(query, callback){
       if(!query.start || !query.end || !query.metric || !query.aggregator){
          throw 'Query parameters missing, min start, end, metric, aggregator';
       }
       //Query creation
       var queryURL = "http://"+this.host+":"+this.port+"/api/query?
start="+query.start+"&end="+query.end+"&m="+query.aggregator+":"+query.metric
;

       if(query.tags){
       queryURL += '{';
          var ntags = Object.keys(query.tags).length;
          var j = 1;
```

```
            for(i in query.tags){
               if(j != ntags){
                  queryURL += i+'='+query.tags[i]+",";
               }else{
                  queryURL += i+'='+query.tags[i];
               }
               j++;
            }
         queryURL += '}';
         }else{
            queryURL +='{}';
         }
        //Query correctly created
        http.get(queryURL, function(ress) {
        //http request to opentsdb
           var responseParts="";
           ress.on('data', function (chunk) {
               responseParts+=chunk;
           });
           ress.on('end', function () {
            callback(responseParts);
           });

         }).on('error', function(e) {
            callback();
         });
      }
   }
```

Finally as any node module we have to export it.

```
module.exports = Nodetsdb;
```

3.3.2 Client - OpenTSDB connector (nodetsdb-client)

nodetsdb-client differs the most among all of the connectors. In the others, the node server is always in the middle of the transaction of the data points. With this connector, all the work is on the client. In **Figure 3.14** we can observe how the server only has to create and serve the final HTML with all the javascripts. One of those javascripts, plots.js has the code to query and retrieve the data points from OpenTSDB directly. In the same figure we can see how the system is organised based on visual analytics sections.

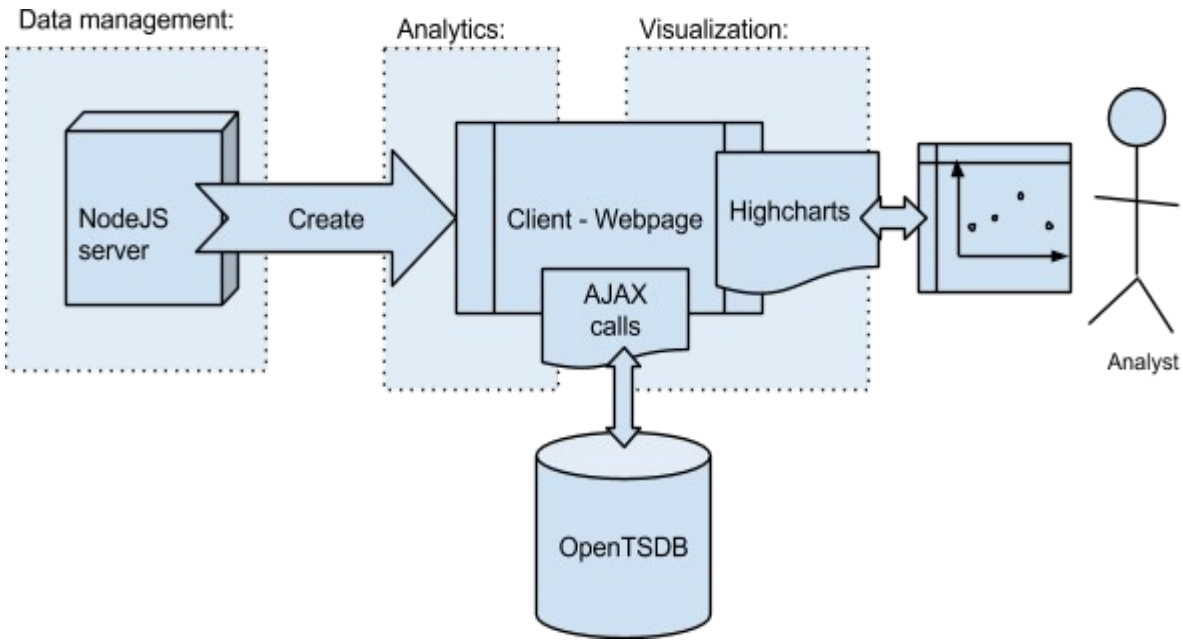


Figure 3.14: Structure of the nodetsdb connector, showing the visual analytics sections

The main advantage of this method is that the server workload will be low even if there is a large amount of clients accessing at the same time. On the other hand, all of the processing is in the javascript of the client, depends on the client's machine performance, and needs direct access to OpenTSDB. It also has a limitation in the amount of data transformation you want to include- heavy transformations will not be possible to do in the client.

The best use of this connector could be in situations where there is not a security concern in having a direct connection with OpenTSDB and when the amount of data points requested are not excessively big.

3.3.2.1 nodetsdb-client

nodetsdb-client is a JavaScript library that is an adaptation of the nodetsdb module from node. Its code is similar, with the difference existing in how the final request to OpenTSDB is executed. In the case of nodetsdb, it was done with the HTTP module of node and in this case, it is done with an AJAX request with the jQuery library.

nodetsdb-client is also hosted in GitHub[36] and released under an open source BSD Clause-3 license.

It is used in the same way as nodetsdb, but in this case we use it in the public javascript file plots.js

```
function handleClient(amountPoints){
        var options = getQueryData(amountPoints);
        var nodetsdb = new Nodetsdb({host:'opentsdbserver.com', port:4242});
        nodetsdb.getDataPoints(options, function(data){
                        plot_data(data,true);
                });
};
```

The source is linked in the layout template as the file nodetsdbclient.js. Below, we compare nodetsdb and nodetsdb-client. Wich main difference is the library used to make the HTTP request, a node library in the nodetsdb and a function of jQuery in nodetsdb-client.

```
//nodetsdb-client
$.ajax({
                url: queryURL,
                jsonp: "jsonp",
                dataType: "jsonp",
                success:
function( response ) {


        callback(response);
                                }
        });
```

```
//nodetsdb
http.get(queryURL, function(ress) {
        var responseParts="";
          ress.on('data', function (chunk)
{
              responseParts+=chunk;
        });
        ress.on('end', function () {
          callback(responseParts);
        });

      }).on('error', function(e) {
         callback();
      });
```

3.3.3 Python connector

The Python connector is created so Python code could be used to transform and manage data from OpenTSDB. Python is a popular and efficient programming language. It is ideal to use it to create functions to manage data points. The motivation of using this connector was the experience with python of some actual users of OpenTSDB that managed to create scripts for accessing and managing data points from OpenTSDB.

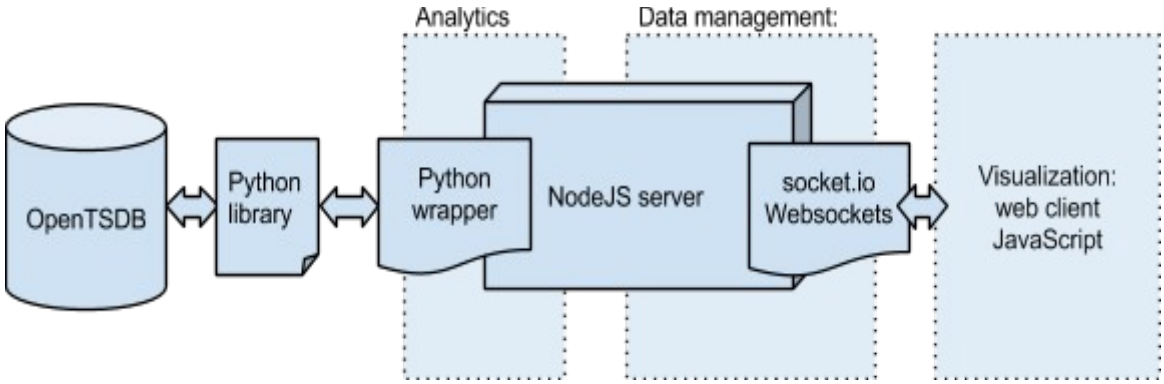


Figure 3.15: Structure of the Python connector, showing the visual analytics sections

For this connector we used as main Python script the project opentsdb_pandas hosted on GitHub[37] made by the supervisor of the thesis, Tomasz Wiktor Włodarczyk. We created a python wrapper in JavaScript that directly executes the a python script that uses the library mentioned above. The complete structure of the connector is shown in the **Figure 3.15.**

For this connector we opted for a more direct connection, executing the python script directly from node instead of a more modular and complex connection, such as the one explained in the next section.

To use this connector we will receive a request through websockets and then, the controller (app.js) will proceed to execute the python script.

```
/* Python Mode */
socket.on('getDPPythonMode', function (options) {
      var python = require('child_process').spawn('python',
        // second argument is array of parameters
        ["./python_files/entryPoint.py"]
          );
          var output = "";
          python.stdout.on('data', function(data){ output += data });
          python.on('close', function(code){
                  if (code !== 0) {
                     throw 'Problem executing Python script';

                  }else{
                           socket.emit("dataServerPython",output);
```

```
                }
        });
});
```

The python script mentioned in the code will use the opentsd_pandas library to fetch the data points of OpenTSDB. For testing purposes the query is fixed in the script. The python script will print the points to the standard output in a JSON format that the node server will emit through the websocket to the client. Then the client will finally parse the JSON and plot it with Highcharts.

```
import opentsdb_pandas as opd
import datetime as dt
import urllib2
import json

ts1 = opd.ts_get('cipsi.weather.TA', dt.datetime(2014, 4, 4, 12, 00),
dt.datetime(2014, 4, 13, 12, 00), 'station=44640',
hostname='haisen36.ux.uis.no')

aux=json.dumps(ts1.T.as_matrix().tolist(),indent=4)
print aux
```

Besides the advantages already mentioned about fetching the data from the server, this connector will allow Python to apply any transformation to the data. But because of the direct implementation of the connector, any Python script needs to meet the same requirements like printing the output in a JSON format.

3.3.4 R connector

This connector is based on the idea of using the programming language R to fetch and manage the data before sending it to the node server. We choose R because it has very useful built-in functions to transform data. We opted for a more complex but modular structure that will allow for easy addition of new R functions in the future.

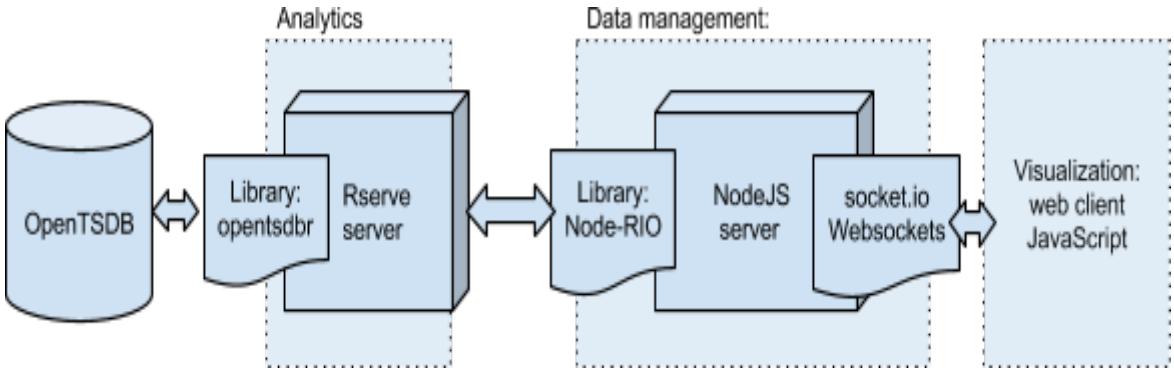


Figure 3.16: Structure of the R connector, showing the visual analytics sections

In the current implementation, shown in **Figure 3.16,** we only use it to fetch data from the OpenTSDB without applying any algorithms, but in further implementations we will use predictions and error detection algorithms as well as statistical analysis.

To connect to OpenTSDB with R we have two libraries: opentsdbr[38] and R2Time[39]. Opentsdbr is a simple library that will allow us to only read values in OpenTSDB with R code. It uses HTTP for the request, making it very inefficient, but simple to use. Additionally, we have R2Time- a library developed by Bikash Agrawal, a PhD student at the University of Stavanger. It is a more optimized and complex library that goes directly to HBase to fetch the data. For the purpose of building the first version of the solution, we will use opentsdbr.

We need to serve the data obtained in R to the node server. To do so, we will use the R library Rserve, a library that creates a TCP/IP server in R so that the data management layer can make queries with R code. The node server will send requests to Rserve and it will translate those requests in R code, execute them, and give back the results.
In order to communicate Rserve with the node server, we need an Rserve client built-in javascript and made for Node. There are three node modules that meet our requirements: Rserve-client[39], Rserve-js[40], and Node-RIO. Rserve-client meets the requirements, however it cannot retrieve plots directly from Rserve. Moreover, the last activity in the code was from five months ago and it would be unwise to trust an abandoned library. Another viable option is Rserve-js, a library that has more functionality than Rserve-client, yet it does not integrate well with error handling and

is not optimized for managing large amount of data. Finally, Node-RIO is a complete library and compensates for all of Rserve-js's issues. It can work with error handling and is optimized for speed in data transfer. The three libraries operate on the same principle, fetching points from Rserve and converting them in a JSON object so they can be manipulated in javascript easily. For our solution, we have chosen Node-RIO because of its performance and error handling. We then execute Node-RIO when receiving a request from the client through the websockets.

```
//app.js
// R mode
  socket.on('getDataPoints', function (options) {
    var data = {
      metric: 'cipsi.seeds.test1.temperature',
      start: {timestamp:'2013-08-04 12:00:00', timezone:'CEST'},
      end: {timestamp:'2013-08-07 14:00:00', timezone:'CEST'},
      tags:[{name:'node', value:'0013A2004061646F'}],
      debug:true
    }

    executeRio(data,function(result){
      socket.emit("dataServer",result); });
    });
```

One very good advantage is the modularization, it is really easy to add new functions in R a programming language specialised in statistical analysis. But also this is it is great disadvantage, in order to gain modularization, we increased its complexity. This complexity will pay off later when managing a large amount of data points.

3.4 General workflow

In the current implementation of our node server, we are not going to implement the reactive workflow explained in visual analytics because of its complexity. In the future, when we have added functionality like error detection and notifications, the reactive workflow will run in the background monitoring the dataset and analysing it to detect errors and if so, notifying the user.

The interactive workflow is completely integrated and implemented in our system. It is the interaction of an analyst with the system. When the analyst requests the main page, the interaction activates the workflow in the server to gather the data and send it back to the analyst. There are two different groups of interactive workflows, the one that includes the server to request the data points (nodetsdb, python or R connectors) or the one in which the client handles the connection with OpenTSDB (nodetsdb-client).

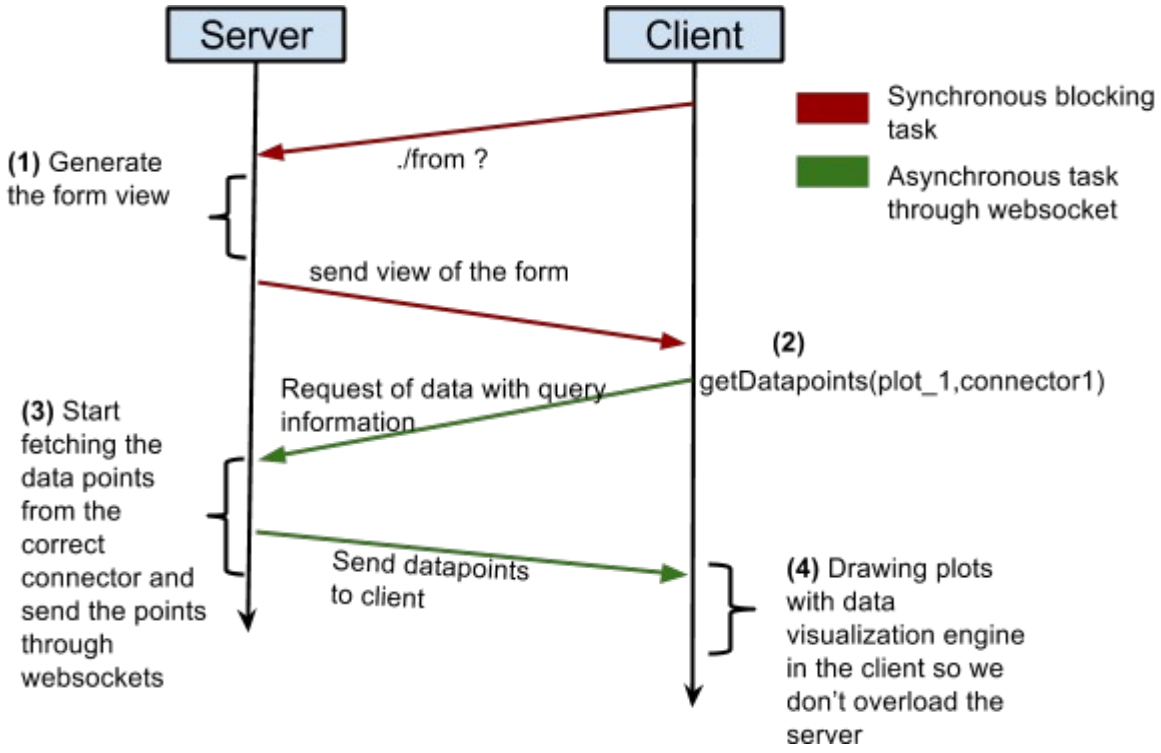


Figure 3.17: Interactive workflow for nodetsdb, python, and R connectors

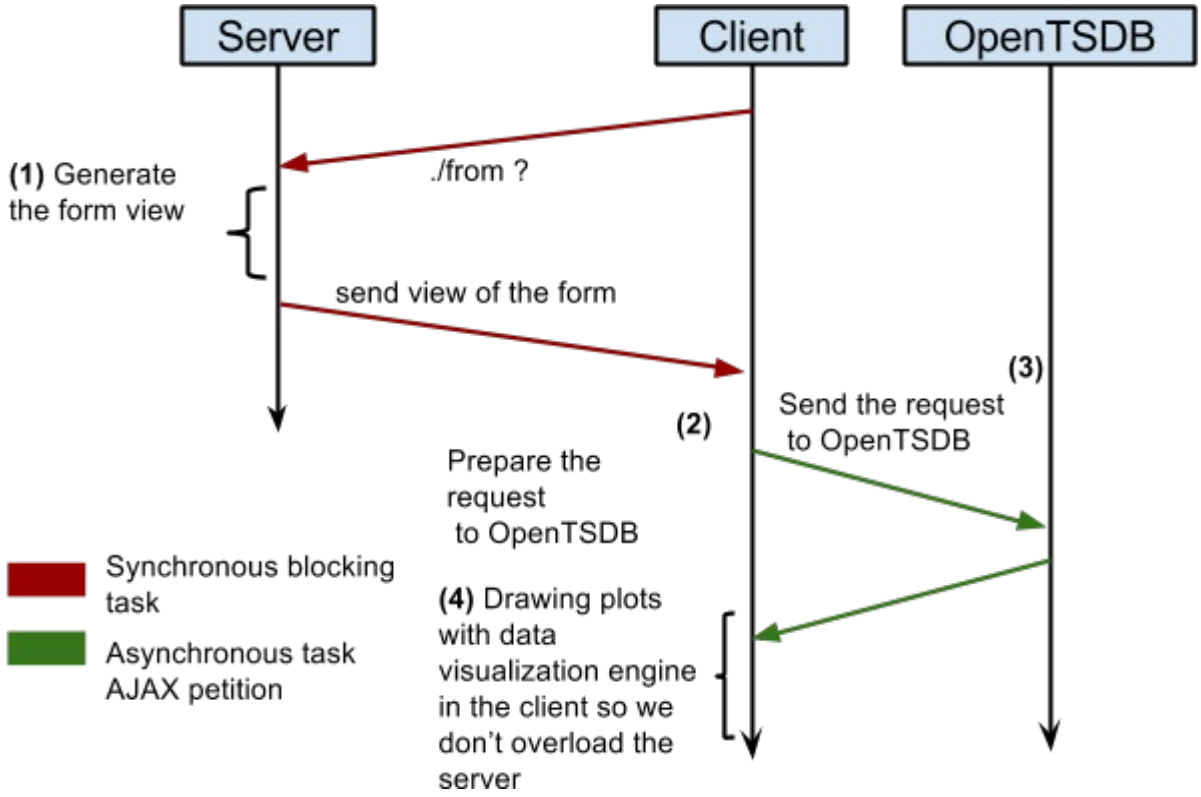


Figure 3.18: Interactive workflow for nodetsdb-client

In point (1) of **Figure 3.17**, the server sends empty form and the client asks through websockets to get the data points of a specific query. The server then queries to OpenTSDB with the specific connector. Once the server has the data, it is sent through the same websocket of the request. One of the advantages of using NodeJS and websockets is that they are not blocking and we can put a progress bar per plot or another mechanism to minimize the impact of waiting for the results.

On the other workflow, **Figure 3.18** the server sends the empty form and then the client takes care of querying OpenTSDB once the form is done. No websockets are involved and, when the request is done, it plots the data points from the same javascript that did the query.

4 Results & Discussion

In order to test the different connectors, we executed the following four times: three sets of measures with different amounts of data points to compare the connectors and see if there is room for improvement. The dataset used is one that we have in the OpenTSDB managed by the University of Stavanger, and this dataset will be the final database used along the interface.

We created three sets of scenarios (set-1, set-2 and set-3), the difference between them being the amount of data points (different time frames). The first (set-1) from 2014/04/04 12:00:00 to 2014/04/13 12:00:00, the second one (set-2) from 2014/04/04 12:00:00 to 2014/04/21 12:00:00, and the last one (set-3) from 2014/04/04 12:00:00 to 2014/05/06 12:00:00. The total number of data points in the first case is 1284 (set-1), in the second case is 2435 (set-2), and in the third case 4567 (set-3) representing the temperature measure by a sensor. Each test was repeated four times and an average was calculated.

4.1 Timing of NodeJS - OpenTSDB connector (nodetsdb)

The piece of code that we are testing is from the point we create the object nodetsdb to have the points ready to be plotted.

```
//app.js
//Start measuring the time
var nodetsdb = new nodetsdblib({host:config.opentsdbserver,
port:config.opentsdbserverport});

        nodetsdb.getDataPoints(options, function(dp){
                //Stop measuring the time
                socket.emit("dataServer",dp);
        });
```

	Test 1	Test 2	Test 3	Test 4	Average
set-3 (4567)	57 ms	58 ms	55 ms	54 ms	56 ms
set-2 (2435)	41 ms	33 ms	38 ms	38 ms	37.5 ms
set-1 (1284)	25 ms	27 ms	25 ms	24 ms	25.25 ms

Table 4.1: Time in ms of the nodetsdb module.

Knowing that nodetsdb relays on an HTTP request to obtain data, these times show that obtaining data from a month, taken every 10 min 24/7 (set 3) is fast, even if we wanted to show data from a year, in the same conditions, the transaction would not even be one second (a rough approximation of 0.675 sec).

4.2 Timing of Client - OpenTSDB connector (nodetsdb-client)

We are going to measure the time spent by the library nodetsdb-client to fetch data from OpenTSDB in the client, using Google Chrome browser Version 35.0.1916.114. In the following piece of code, we see exactly where the measurements are taken.

```
//plots.js
//Stat timing
var nodetsdb = new Nodetsdb({host:'haisen36.ux.uis.no', port:4242});
        nodetsdb.getDataPoints(options, function(data){
                                //End timing
                               plot_data(data,true);
                });
```

	Test 1	Test 2	Test 3	Test 4	Average
set-3 (4567)	89 ms	81 ms	80 ms	82 ms	83 ms
set-2 (2435)	58 ms	60 ms	55 ms	52 ms	56.25 ms
set-1 (1284)	46 ms	42 ms	40 ms	39 ms	41.75 ms

Table 4.2: Time in ms of the nodetsdb-client library.

These times are not as good as the previous connector, but is also fast for the amount of points requested- only 84 ms for a month's worth of data. The main reason for the difference will be the optimization performance of nodejs versus the javascript in the client. In any case, it is a very suitable connector for direct access to OpenTSDB.

4.3 Timing of Python connector

For the Python connector we made two different measures, one in the javascript code from the script to be executed to when the points are available. For the second

measurement we used the unix command “time” that counts the time that the system takes to execute a specific script.

The code mentioned before is shown above.

```
//app.js
//Start timing
var python = require('child_process').spawn(
        'python',
        // second argument is array of parameters
        ["./python_files/entryPoint.py"]
        );
        var output = "";
        python.stdout.on('data', function(data){ output += data });
        python.on('close', function(code){
                //Timing end
                socket.emit("dataServerPython",output);
            }
        });
```

	Test 1	Test 2	Test 3	Test 4	Average
set-3 (4567)	379 ms	371 ms	375 ms	387 ms	378 ms
set-2 (2435)	336 ms	345 ms	372 ms	330 ms	345.75 ms
set-1 (1284)	325 ms	313 ms	320 ms	324 ms	320.5 ms

Table 4.3: Time of requesting points and getting them (Node-python wrapper + python script)

	Test 1	Test 2	Test 3	Test 4	Average
set-3 (4567)	450 ms	426 ms	443 ms	438 ms	439.25
set-2 (2435)	340 ms	336 ms	349 ms	344 ms	342.25
set-1 (1284)	327 ms	323 ms	318 ms	342 ms	327.5

Table 4.4: Time of requesting points and getting them (python script alone, using "time" command)

We can see a visual comparison of the averages of both timings in Figure 4.1.

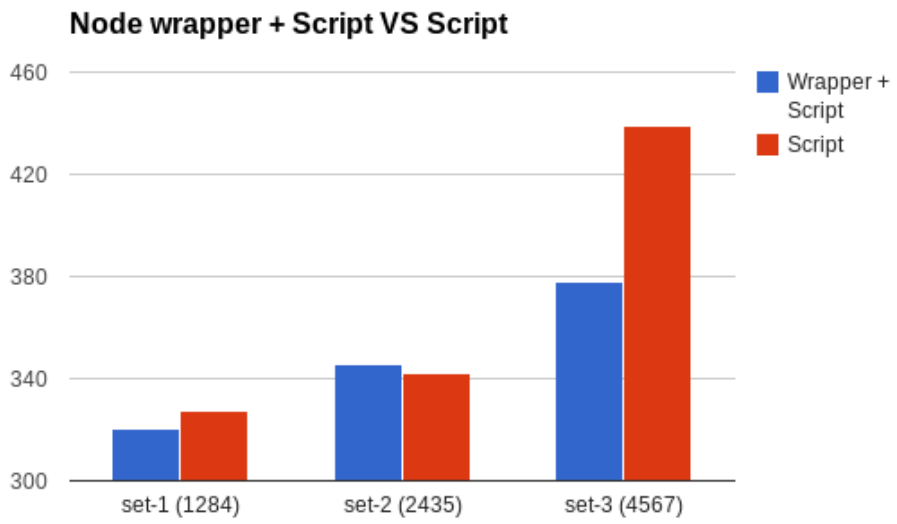


Figure 4.1: Comparison between the JavaScript wrapper and only the script.

At first sight it may not make sense that the script alone takes more time than the script and the wrapper, but we have to remember that the script prints in the standard output. When using the JavaScript wrapper, we catch the standard output into a variable- but, when executing the unix function time, it has to print in the terminal all of the data points in JSON format. This process is more time consuming than printing the JSON then catching it on a variable.

Comparing these timings with the previous connector, it is around 5 times slower than the nodetsdb connector, mainly because of data flow, that start OpenTSDB - Python and then Python – NodeJS. The result being two steps that in nodetsdb is one. Also, the transformation of the raw data into Python structure and then again, transforming it to JSON to send to nodetsdb contributes to the slower time.

Even though the time is worse than the others, it is still fast enough to be usable in many situations where Python is required or preferred to manage data.

4.4 Timing of R connector

For the R connector we measure the total time used by the node server to obtain all the points and get them ready to be sent to the client. Because this connector is more complex than others we will expect higher times.

```
//app.js
// R mode
  socket.on('getDataPoints', function (options) {
    //Start timing
    executeRio(data,function(result){
       //End timing
       socket.emit("dataServer",result); });
    });
```

It seems like a short piece of code but the call executeRio will make an HTTP request to Rserve that will execute a script containing the library opentsdbr that will query OpenTSDB, and then the data points will do the same journey backwards.

In addition to the measures, we also time which part of the first time the R library opentsdbr takes.

	Test 1	Test 2	Test 3	Test 4	Average
set-3 (4567)	4089 ms	4095 ms	4496 ms	4400 ms	4270 ms
set-2 (2435)	2753 ms	2689 ms	2672 ms	2718 ms	2708 ms
set-1 (1284)	1884 ms	1867 ms	1901 ms	1931 ms	1895.75 ms

Table 4.5: Time of requesting points and getting them with the R connector

	Test 1	Test 2	Test 3	Test 4	Average
set-3 (4567)	3239 ms	3221 ms	3587 ms	3574 ms	3405.25
set-2 (2435)	1666 ms	1599 ms	1623 ms	1646 ms	1633.5
set-1 (1284)	670 ms	675 ms	732 ms	683 ms	690

Table 4.6: Time used by opentsdbr to query OpenTSDB and get the results back

In **Figure 4.2** we can observe the relation between both.

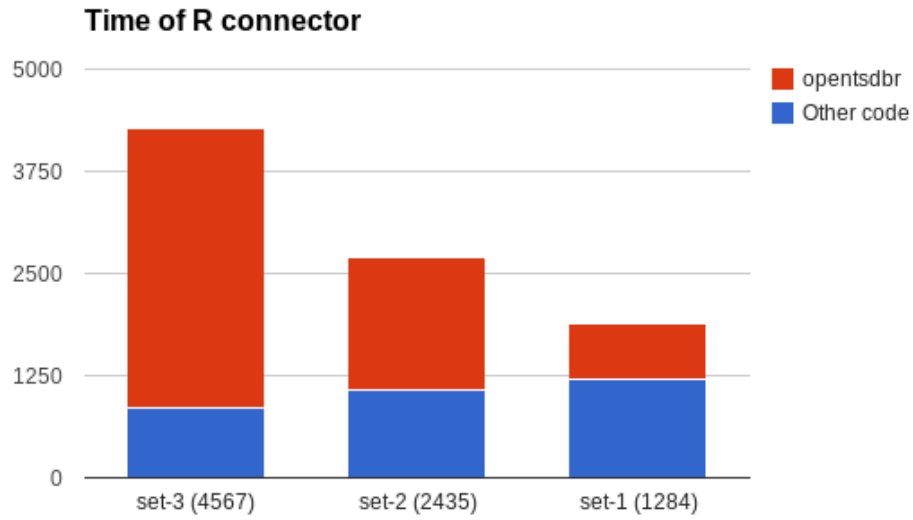


Figure 4.2: Timing of the R connector by sections

We can observe in **Figure 4.2** that most of the time taken by the R connector is used by the library opentsdbr, showing us that there is room for future improvement.

With these results we can conclude that the library opentsdbr is the bottleneck of the whole system and in future developments we will test the other library R2Time to see its performance under the same circumstances.

4.5 Timing comparison

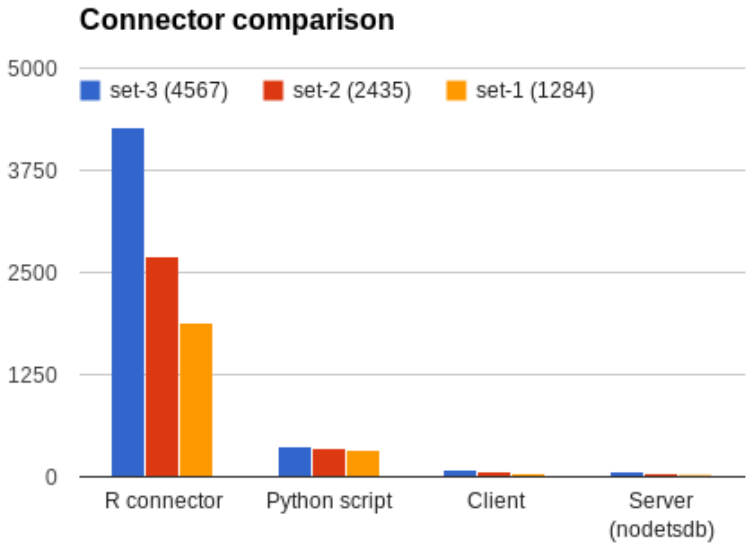


Figure 4.3: Time comparison between connectors

As we can see in **Table 4.5,** the average time to obtain 4567 data points takes around 4,5 seconds, a huge amount of time compared with the other connectors that do not get to half a second for the same amount of points. The difference can be observed in **Figure 4.3** as well.

This huge increase in time in the R connector is because of the modularized architecture of the R connector, increasing its complexity. For future versions it will be recommended to compare with a more direct approach like the python connector or change the R library used. This specific architecture of the R connector is not suitable to a real environment because of its great delay.

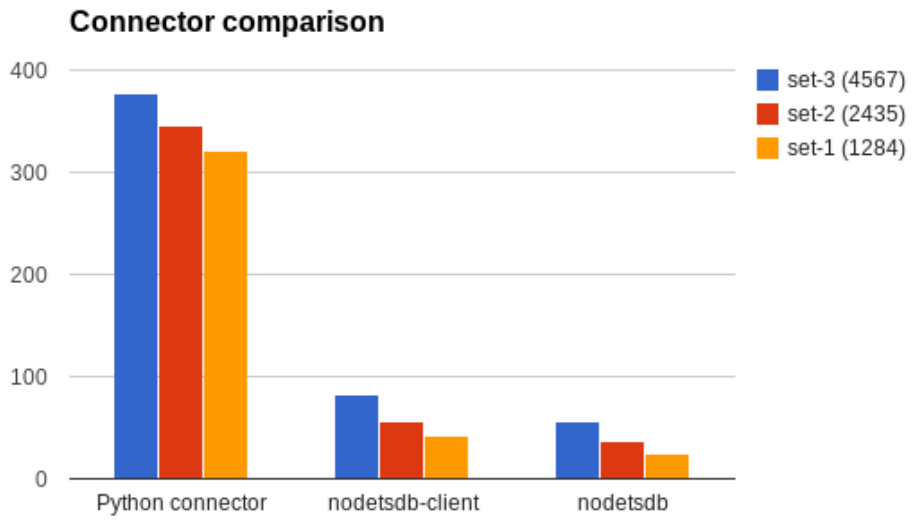


Figure 4.4: Python, nodetsdb, and nodetsdb-client compared

If we take a closer look at data without the R connector, shown in **Figure 4.4** we can get a better idea of the other connector performances. It is clear that nodetsdb performs better and is recommended to be used for a day-by-day connector. While nodetsdb-client performs very well, it needs the client to have direct communication with OpenTSDB and sometimes that cannot be granted. Even when the python connector is not as fast as nodetsdb, it could be a good alternative when it is necessary to apply transformations to large amounts of data.

5 Conclusions

openTSDBnode is a scalable and user friendly dashboard that implements four different connections with different programming languages that will enable data manipulation in any of the languages mentioned above.

As the results shows there is room for improvement, the connector that needs a more deep work is clearly the R connector. The initial idea of busing the opentsdbr library lead us to awful result. As shown in the results only changing that library will improve the whole connector.

openTSDBnode as well as the main libraries are released to the community with a BSD 3-Clause License, to allow the community to modify and improve the work done here.

5.1 Future work

As mentioned above, one of the first tasks for future work will be changing the R library, opentsdbr, for a faster and more efficient one. There are alternatives, such as R2Time, that are worth trying.

The next main task to improve opentsdbnode will be to optimize and modularize the Python connector, the actual implementation is a proof of work, but needs to be refined.

6 References

[1] "OpenTSDB - A Distributed, Scalable Monitoring System." 2010. 11 Dec. 2013 <http://opentsdb.net/>

[2] "HBase - Apache HBase™ Home." 2010. 11 Dec. 2013 <http://hbase.apache.org/>

[3] "HTTP API — OpenTDSB 2.0 documentation." 2011. 21 Jun. 2014 <http://opentsdb.net/http-api.html>

[4] Thomas, Jim, and Pak Chung Wong. "Visual analytics." *IEEE Computer Graphics and Applications* 24.5 (2004): 0020-21.

[5] Thomas, James J., and Kristin A. Cook, eds. "Illuminating the path: The research and development agenda for visual analytics." (2005).

[6] Fekete, Jean-Daniel. "Visual Analytics Infrastructures: From Data Management to Exploration." *Computer* 46.7 (2013): 22-29.

[7] "node.js." 2009. 14 Dec. 2013 <http://nodejs.org/>

[8] "Benchmarking Node.js - basic performance tests against Apache + ..." 2011. 14 Dec. 2013 <http://zgadzaj.com/benchmarking-nodejs-basic-performance-tests-against-apache-php>

[9] "Projects, Applications, and Companies Using Node · joyent ... - GitHub." 2011. 14 Dec. 2013 <https://github.com/joyent/node/wiki/Projects,-Applications,-and-Companies-Using-Node>

[10] "Express - node.js web application framework." 2010. 21 Jun. 2014 <http://expressjs.com/>

[11] "Hogan.js." 2013. 21 Jun. 2014 <http://twitter.github.io/hogan.js/>

[12] "{{ mustache }} - GitHub Pages." 2013. 21 Jun. 2014 <http://mustache.github.io/>

[13] "vol4ok/hogan-express · GitHub." 2012. 21 Jun. 2014 <https://github.com/vol4ok/hogan-express>

[14] "WebSocket.org | The Benefits of WebSocket." 2011. 21 Jun. 2014 <http://www.websocket.org/quantum.html>

[15] "ajax - WebSockets protocol vs HTTP - Stack Overflow." 2013. 21 Jun. 2014 <http://stackoverflow.com/questions/14703627/websockets-protocol-vs-http>

[16] "Grunt: The JavaScript Task Runner." 2012. 21 Jun. 2014 <http://gruntjs.com/>

[17] "MongoDB." 2008. 21 Jun. 2014 <http://www.mongodb.org/>

[18] "jQuery." 2006. 21 Jun. 2014 <http://jquery.com/>

[19] "Bootstrap." 2008. 21 Jun. 2014 <http://getbootstrap.com/>

[20] "Highcharts - Interactive JavaScript charts for your webpage." 2007. 21 Jun. 2014 <http://www.highcharts.com/>

[21] "Welcome to Python.org." 2006. 21 Jun. 2014 <https://www.python.org/>

[22] "The R Project for Statistical Computing." 21 Jun. 2014 <http://www.r-project.org/>

[23] "holstius/opentsdbr · GitHub." 2013. 21 Jun. 2014 <https://github.com/holstius/opentsdbr>

[24] Urbanek, Simon. "Rserve--A Fast Way to Provide R Functionality to Applications." *PROC. OF THE 3RD INTERNATIONAL WORKSHOP ON DISTRIBUTED STATISTICAL COMPUTING (DSC 2003), ISSN 1609-395X, EDS.: KURT HORNIK, FRIEDRICH LEISCH & ACHIM ZEILEIS, 2003 (HTTP://ROSUDA. ORG/RSERVE* 2003.

[25] "albertosantini/node-rio · GitHub." 2011. 21 Jun. 2014 <https://github.com/albertosantini/node-rio>

[26] "gRaphaël—Charting JavaScript Library." 2009. 16 Dec. 2013 <http://g.raphaeljs.com/>

[27] "JavaScript InfoVis Toolkit - Nicolas Garcia Belmonte." 2013. 16 Dec. 2013 <http://philogb.github.io/jit/>

[28] "MooTools Forge | MilkChart." 2009. 16 Dec. 2013 <http://mootools.net/forge/p/milkchart>

[29] "Update to jQuery Visualize: Accessible Charts with HTML5 from ..." 2010. 16 Dec. 2013 <http://filamentgroup.com/lab/update_to_jquery_visualize_accessible_charts_with_html5_from_designing_with/>

[30] "moochart - charts for mootools." 2008. 16 Dec. 2013 <http://moochart.coneri.se/>

[31] "JS Charts - Free JavaScript charts." 2007. 16 Dec. 2013 <http://www.jscharts.com/>

[32] "SIMILE Widgets | Timeline." 2009. 16 Dec. 2013 <http://www.simile-widgets.org/timeline/>

[33] "D3.js - Data-Driven Documents." 2010. 16 Dec. 2013 <http://d3js.org/>

[34] "BobString/nodetsdb · GitHub." 2014. 22 Jun. 2014 <https://github.com/BobString/nodetsdb>

[35] "nodetsdb - npm." 2014. 22 Jun. 2014 <https://www.npmjs.org/package/nodetsdb>

[36] "BobString/nodetsdb-client · GitHub." 2014. 22 Jun. 2014 <https://github.com/BobString/nodetsdb-client>

[37] "twwlodarczyk/opentsdb_pandas · GitHub." 2014. 22 Jun. 2014 <https://github.com/twwlodarczyk/opentsdb_pandas>

[38] "holstius/opentsdbr · GitHub." 2013. 14 Dec. 2013 <https://github.com/holstius/opentsdbr>

[39] "rserve-client - npm." 2012. 18 Dec. 2013 <https://npmjs.org/package/rserve-client>

[40] "cscheid/rserve-js · GitHub." 2013. 17 Dec. 2013 <https://github.com/cscheid/rserve-js>