Elastic Rapid Provisioning, Multiple Source Monitoring Framework Architecture

Akshay Potnis
Dept. of Computer and Information Technology, PVG's COET, University of Pune, India

Rohan Khadilkar
Dept. of Computer and Information Technology, PVG's COET, University of Pune, India

Sarang Rakhecha
Dept. of Computer and Information Technology, PVG's COET, University of Pune, India

Vallabh Naik
Dept. of Computer and Information Technology, PVG's COET, University of Pune, India

ABSTRACT
In this paper, elastic, multiple sources monitoring framework architecture which can be rapidly provisioned with respect to the monitoring requirements is being proposed. The need of system performance monitoring is of prime concern along with a tool to monitor user related performance i.e. in case of a product firm; they might need to monitor the sales regularly to predict some future trends. The system metrics combined with logs need to be plotted side by side to extract the similarity between them to predict efficiency of system resource usage. Moreover, each user might be using a different database as data source .Bearing this multi-faceted heterogeneity in mind, framework architecture for multiple-source, multipurpose monitoring is being proposed which can give the user a fully satisfying monitoring experience. A new concept of "MVC as an algorithm" can be an accurate measure for an efficient cloud based monitoring service (SAAS) and can also be incorporated with the framework architecture.

General Terms
Cloud Computing, Resource Monitoring

Keywords
Software as a Service (SAAS), Graphing-mechanism, JavaScript Object Notation (JSON), Monitoring, Parsing, Time-series database, Web services

1. INTRODUCTION
Trending is the practice of collecting information and attempting to spot a pattern, or trend, in the information. Monitoring is the process of collecting the data combined with trending. This is a very crucial part of every testing as well as management team of a company. While the testing team defines monitoring in terms of system resources, the management does the same in terms of returns or sales or profits. In either case, a clear graph of all the requirements seems to fulfill the tasks. But these requirements are subject to change with respect to data storage as well as terms of monitoring. Considering these requirements there is a dire need of a system that can be elastic in terms of any user requirements, be it heterogeneous data sources or be it multiple graphing mechanisms. Hence, an architecture that can provide elasticity in terms of all the factors from User Interface to Data Source is necessary. This can cater to all requirements from a single place, with the client just needing to choose his monitoring requirements and use the service. Furthermore, this architecture can be a perfect model for Software as a Service (SAAS)[1] that can serve any kind of monitoring requirements.

2. ARCHITECTURE AND DESIGN
The architecture is comprised of five layers namely Client Layer, UI management Layer, Routing layer, Web Services Layer and Data Source Layer. The first and the last

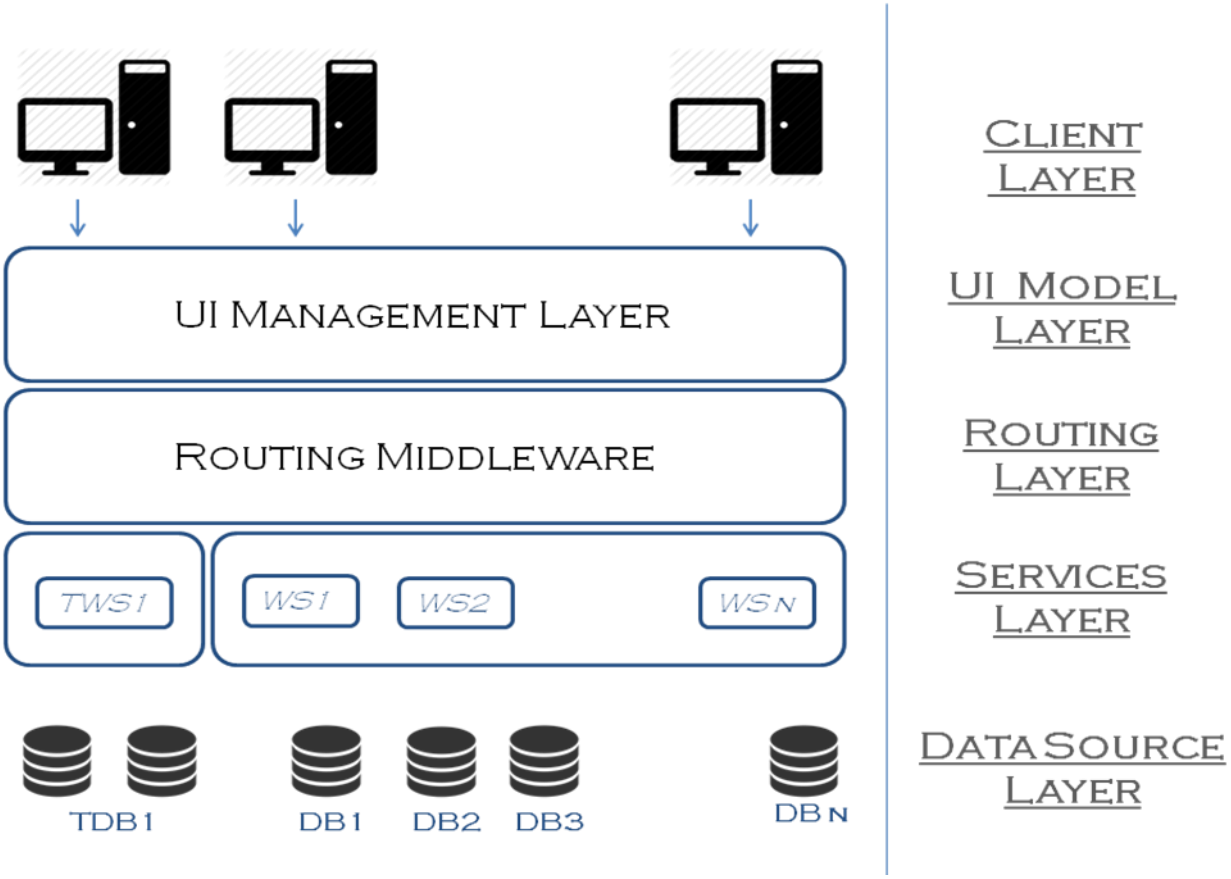

Fig.1 Proposed Architecture Design (Layer-wise)

layer may or may not be in one place i.e. may be a part of a distributed system or a part of a virtual private network. So this favors the mapping between client computers to data sources to be an n to n mapping. Each layer is designed to perform its own functions.

2.1 Client layer

This layer shall be responsible for maintaining the different user interfaces in terms of Monitoring dashboards[2] that the client creates. It shall also hold a copy of Sources. JSON file that shall be a list of all the sources that the current client is monitoring along with the parameters related to those sources. Each client shall have a copy of both these, namely, a folder storing all client dashboards and a sources file. The dashboards folder shall also contain separate JSON files for each dashboard that the user creates. When the client browser requests for the main dashboard, a common dashboard.html file shall be served but with the structure derived from parsing his Dashboard and Sources JSON files. The detailed structure of both these files is explained in the next section. Use of JSON files makes it very easy and efficient in terms of maintenance and efficiency of data exchange and thus makes the front end structure light weight.

2.2 UI management layer

This is the layer that is the most important layer in terms of bringing in the UI based dynamic facilities as well as handling all the business logic related to the manipulation of data and the rendering sequence of the dashboard. This layer shall contain all the common logic needed by all the 'n' clients that shall be using this service. To be specific, it shall contain the dashboard renderer, request query parser, response handler, and data formatter, dashboard saving and loading mechanism and graphing mechanism [3][4]. From this layer starts the centralized part of the architecture. Grouping these facilities together as one common layer reduces the need of maintaining separate copies of these on the client side, which is the case for all the monitoring tools as of now. Moreover, since they have to perform the same function in case of all data sources, keeping them as a common part makes this single layer perform all the tasks that are currently being implemented by four different tools on a single client.

2.3 Routing layer

As the name suggests, this layer shall handle the routing of requests made by the clients. Since the client may be requesting for data from any number of sources, each request from the client must be routed to the specific web service for the requested source. This is a very important task. This layer shall maintain a file named Routes. JSON which shall contain entries of the format:

```
[
        { "source" : "path" },
        { " source" : "path" },.
];
```

Whenever there is a request from the client this file shall be searched for the specific source and the entry for its corresponding path shall be popped out. The request shall then be made to the service located on that path. When a new web service is added to the system, all that needs to be done is to add an entry to the routes file. This layer shall thus contain a routes file and a Route Finder script which finds the path from

this file. Also, a client specified routes file will be provided, whose significance shall be made clear in the next layer. This structure makes routing very easy to maintain and update according to the changes in the system.

2.4 Web services layer

This layer shall be the most open ended layer in the system structure. It shall perform the task of requesting data from heterogeneous data sources and send the obtained data back to the UI management layer. Each web service shall be a REST[5] like service destined to get data from a specific source e.g. WS1 gets data from a Time Series database[6], WS2 gets data from a No SQL database[7] etc. The mapping of WS to source shall be done in routing layer. This can also contain web services for getting data from time series databases that are specially designed for handling data associated with system monitoring. Each web service shall also be following an ordered set of steps in terms of their working mechanism. These steps should be strictly followed by the client to create a personalized web service in case of a specific data source requirement. The next part comprises of addition of an entry to this file, mentioning the data source name and the path to the web service. A corresponding entry also needs to be added to Sources. JSON file in the client layer for making this service available. Moreover, there is no restriction on programming language as long as the steps are followed and data is returned in the specified format. Thus, this structure shall favor serving multiple sources as well as addition of client specific sources when the need arises.

2.5 Data sources layer

Each client data source might be in different places and in different formats. It is the job of the web service to request data and provide it to the UI management layer to do the further execution. The security related specifics for each data source needs to be mentioned by the client in the params section of the Sources. JSON file. Thus, this layer is a distributed physical layer that shall be representing the source of raw data.

In this architecture, the process of data collection has not been added. Data collection is the foremost step which occurs prior to the actual monitoring. The reason being, data collection is a part of the activities that the client might be doing irrespective of monitoring. It is only in case of system based monitoring that the collection doesn't come as part of the daily data collection activity. For this purpose, the collection has not been considered as a part of this architecture. But in the future scope & extensions section, a note as to how collection can be incorporated as a parallel working activity to this architecture has been made.

3. DATA FORMAT

In this section, the formats, structure of the dashboard as well as sources files which shall help in making clear how these files shall contribute to the dynamic working of the system has been described in detail..

3.1 Sources. JSON

```
[
    {
        "sourceName" : "Mysql",
        "params" :
        [
            "username": "xyz",
            "password": "abc",
            "hosts": "192.168.0.1",
            "database": "mydb"
        ],
    },
    {
        "sourceName" : "SQLite",
         .
         .
         .
         .
    },
    .
    .
    .
    .
]
```

Fig.2 Sources. JSON file format

This file shall be created by default for every client. It shall contain a list of all the sources with params parameter empty. Source name shall point to the value of the actual name of the sources. The params parameter needs to be filled by the client according to his/her db specification. At the end of the file there shall be default source pointing to demo specification. Initially all the sources shall be commented. As per the requirements the user shall uncomment the source he/she wants and fill in the parameters for it. Only those that are uncommented shall be made available to the user as options of data sources for monitoring on the dashboard. Due to this, the user can be made available sources as per his need by just commenting out the part not required or vice-versa.

```
[
    {
        "name": "MyDashboard",
        "description": "",
        "refresh": 500000,
        "metrics": [
            {
                "name": "myGraph",
                "targets": [],
                "renderer": "area",
                "interpolation": "step-before",
                "description": "",
                "height": 240,
                "width": 465,
                "source": "Mysql",
            }
        ],
        "dbparams": {
            "username": "xyz",
            "password": "abc",
            "hosts": "192.168.0.1",
            "database": "mydb"
        },
        "queries": [
            {
                "sql":"select distinct(timestamp) as Date,name as Name,
                       value as Value from metric where name = \"xyz\"",
                "Id": "My_id"
            }
        ]
    }
]
```

Fig.3 Dashboards.JSON detailed format

3.2 Dashboards. JSON:

This format shall be common for all the client dashboards. The values of the respective parameters may vary from dashboard to dashboard. The name dashboards. JSON is just a type header for explanation. In actual sense this shall be the name by which the dashboard shall be saved. E.g. if you store this dashboard by the name myDash, then the file shall be myDash. JSON. But to keep things generic we shall refer to this file as dashboards. JSON in the rest of the paper.

```
[
  {
    "name": "Dashboard Name",
    "description": "",
    "refresh": 100000,
    "metrics": [],
    "dbparams": [],
    "queries": [],
  }
]
```

Fig.4 Dashboards.JSON file format

This is the default format. These parameters shall be for the dashboard as a whole. Since the dashboard needs to be refreshed when new data arrives, a parameter called refresh rate is added. The metrics section shall contain all the graph related information. To relieve the parsing of sources.JSON once a dashboard is saved, the data source parameters related to that dashboard get saved in dbparams. A special provision for queries is kept if the user wants to retrieve data using his specific query e.g. instead of a single value, an average of the values is needed. But again the query should be according to the required format. An example format for timeseries related values can be:

"select @yourTimestampColName as Date, @yourMetricColumnName as Name , @yourValue as Value from @yourTableName where @yourconditions and @yourTimestampCol between @starttime and @endtime"

Where all parameters with @ are user defined and start time and end time are retrieved from the time resolution chosen by the user for the dashboard.

These queries can be made available to the user along with default query when the user adds targets for graphing to a graph. The detailed format is given in Fig 3.

4. DATA BINDING

Before the actual explanation of the working model, a primary part needs to be mentioned which is 'the dynamic quality of dashboards'. The most important thing in case of monitoring is that requirements always change. Due to this, the dashboard itself needs to be dynamically changeable at any point of time. The dashboards object parsed from the dashboards.JSON file proves to be the basis of this change. When the dashboards.JSON is first parsed, the object can be maintained as a JavaScript object. Since, it is this object that determines the properties of the dashboard structure after rendering, the dashboard can be dynamically changed by just changing the property values of this object and rendering the dashboard again. The diagram Fig 5 may help explain this in a better way.

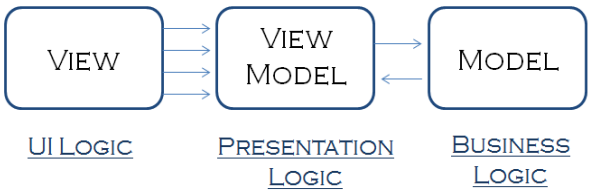

Fig.5 Model View Controller Working diagram

The diagram in Fig 5 represents a general way in which MVC framework [8] actually gets its work done. Also, "MVC as an algorithm" can be incorporated in dashboard parsing. This can be explained with respect to the above diagram. The View represents the dashboard on the browser; the view model represents the UI based functions while Model is the dashboards object. Thus, if the Model is changed, then due to the logical data binding between model, View Model and View, the changes get propagated to the view too. This concept shall be actively responsible for dynamically changing the view at any point of time. E.g. deletion of a graph shall require manipulating the dashboards object and deleting the data of the current graph from the object and rendering the dashboard again.

5. WORKING

This section serves as an explanation for the working of this prototype architecture in the form of an algorithm that shall define the procedural flow of data, requests and responses from one layer to another. When the user opens his dashboard on the browser, the following algorithm shall be applied:

Algorithm:

1. Parse the dashboards JSON file.

2. For every graph i in the metrics section

- Create the graph template from the pre compiled template.
- Find the source mapping for the graph source from the dbparams section. if not mentioned then find mapping from sources.JSON.
- Using the parameters received create a request for getting the data for the graph
- Find the route for the web service using routes.JSON.
- Send this asynchronous request to corresponding web service.
- After response is received, pass on the response data to the data formatter which shall format the data in a specific format required by graphing library.
- Send data to graphing library and create the graph.
- Add graph to the template created in the first step.
- Repeat for each graph.

3. Render the dashboard.

In case the dashboard is already open, then addition and deletion of graphs, saving and retrieving dashboards are some of the activities that need to be elaborated upon. The addition of graphs can be done by manipulating the dashboards object by creating a new graph object according to the properties input by user and appending it to the metrics section of the dashboard object and rendering the page again. Saving of dashboards can be done by just saving the current state of the object into a file with the extension .JSON in the Saved folder with the name input by the user. Loading a dashboard shall be getting a file from the users' Saved folder and replacing the current state of dashboard object by the object formed by parsing the newly acquired JSON file. Due to the concept of MVC applied as an algorithm, anything related to the dashboard dynamics can be done by manipulating the dashboards object. In this way, working becomes both, simplified and easy to understand. It can be extended by anyone as per their requirements.

6. ARCHITECTURE AS CLOUD BASED SERVICE (SAAS)

Monitoring in its purest form is a very dynamic and fluctuating process. It changes with time. Also, at a given time, the needs might be both; systems oriented monitoring of resource usage as well as all other types of monitoring.. In addition, choices of graphing libraries also vary according to personal use. Currently there are many tools serving this purpose, but as the requirements change, the tool needs to change. Buying a new tool with the change in requirements or keeping multiple tools to serve multiple purposes is a very inefficient process. All these problems can be catered to if this

service could be made a cloud based service implemented as SAAS. The architecture perfectly fits this requirement. It can be delivered to users as per their need in the form of Service Level Agreements (SLA's)[9]. Moreover, due to the flow and data exchange efficiency, this architecture also proves a more useful solution than most of the current monitoring tools taken as a whole.

7. FUTURE AND EXTENSION

The most important future scope can be implementing this architecture as a SAAS that can provide the users with a variety of options to serve all monitoring needs. We have also mentioned in the paper about data collection not being a part of this architecture. Another future extension can be data collection mechanism incorporated on the same level as Client layer connected to the Data source layer through the routing layer. This can be a separate protocol stack working side by side with our current architecture. The Collection mechanism can be made a fully functional framework in terms of monitoring requirements. Last but not the least another addition to it can be an data mining system that can help find trends on its own depending upon the data and prompt the results to the user as and when the dashboard is opened. This can prove as a very efficient addition to help finding relation between factors that are least related. This can go a long way in prediction of any kind of future system behavior.

8. CONCLUSION

Although the architecture seems complicated, it's quite easy to be incorporated as a service due to its loosely coupled components and use of data objects as sources of exchange. Furthermore, implementation on the scale of a cloud based service which can prove to be a unique exclusive service that can go a long way in satisfying user requirements at any point of time is very much plausible. Monitoring trends can be focused on more efficiently rather than the time being spent in setting up a new monitoring tool each time a new requirement arises. This service can surely prove a useful means of guessing future trends in the behavior thus minimizing risks, breakdowns as well as costs.

9. REFERENCES

[1] Software as a Service http://www.ibm.com/cloud-computing/in/en/saas.html

[2] Kibana3 Dashboard http://logstash.openstack.org/

[3] JavaScript Graphing Mechanisms or Libraries http://techslides.com/50-javascript-charting-and-graphics-libraries/

[4] Graphing Libraries on Gists Lists https://gist.github.com/eabait/9916975

[5] Representational State Transfer http://rest.elkstein.org/

[6] Time Series Databses https://code.google.com/p/kairosdb/

[7] NoSql Databses http://nosql-database.org/

[8] Model View Controller http://msdn.microsoft.com/en-us/library/ff649643.aspx

[9] Zulkernine, F.H.; Martin, P. "An Adaptive and Intelligent SLA Negotiation System for Web Services", Services Computing, IEEE Transactions on, On page(s): 31 - 43 Volume: 4, Issue: 1, Jan.-March 2011