

Monitoring and controlling ATLAS data management: The Rucio web user interface

M Lassnig¹, T Beermann¹, R Vigne³, M Barisits¹, V Garonne², C Serfon¹, for the ATLAS Collaboration

¹ ATLAS Data Processing, Physics Department, CERN, 1211 Genève 23, Switzerland

² Department of Physics, University of Oslo, 0316 Oslo, Norway

³ Institute for Astro- & Particle Physics, University of Innsbruck, 6020 Innsbruck, Austria

E-mail: Mario.Lassnig@cern.ch, Thomas.Beermann@cern.ch, Ralph.Vigne@cern.ch

Abstract. The monitoring and controlling interfaces of the previous data management system DQ2 followed the evolutionary requirements and needs of the ATLAS collaboration. The new data management system, Rucio, has put in place a redesigned web-based interface based upon the lessons learnt from DQ2, and the increased volume of managed information. This interface encompasses both a monitoring and controlling component, and allows easy integration for user-generated views. The interface follows three design principles. First, the collection and storage of data from internal and external systems is asynchronous to reduce latency. This includes the use of technologies like ActiveMQ or Nagios. Second, analysis of the data into information is done massively parallel due to its volume, using a combined approach with an Oracle database and Hadoop MapReduce. Third, sharing of the information does not distinguish between human or programmatic access, making it easy to access selective parts of the information both in constrained frontends like web-browsers as well as remote services. This contribution will detail the reasons for these principles and the design choices taken. Additionally, the implementation, the interactions with external systems, and an evaluation of the system in production, both from a technological and user perspective, conclude this contribution.

1. Introduction

The high-energy physics experiment *ATLAS* creates non-trivial amounts of data [1]. The data management system *Rucio* [2] catalogues this data and makes it easily accessible for the experiment. The data itself is stored on the Worldwide LHC Computing Grid [3]. Rucio also manages the entire lifecycle of experiment data, from raw detector data up to derived physics data products from user analysis. The governing technical policies are defined by the ATLAS Computing Model [4], and motivate the use of parallel and distributed mechanism to ensure performance and safety of the data.

One of the main components of Rucio is the user interface. There are three interfaces exposed, the REST-based API, the clients software, and the web-based user interface. The REST interface is the lowest level API. Command-line users and external systems mostly interface via the clients software package, currently distributed in Python. The client software uses the REST API underneath. The web-based user-interface, also known as *RucioUI*, gathers to a more user-friendly use case: ATLAS users who want to browse or view information about experiment data, make data transfer requests, view occupancy of storage systems and so forth. It is designed to



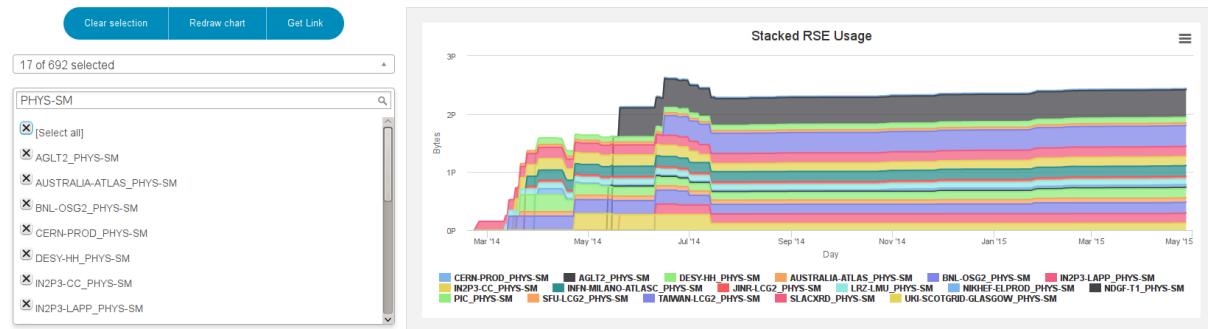


Figure 1. Customisable view of Rucio Storage Element usage.

be as comprehensive as possible for the daily physics analysis tasks, without having to learn intricate data management details.

This paper is structured as follows: first, the architecture of RucioUI is explained, including the web-stack, the security mechanism and the backend. Then, different views available in RucioUI are explained, including, but not limited to, data views, accounting, and the controlling component where users can request replication rules. After a description of the performance monitoring inherent to the RucioUI, the paper closes with a short summary and future outlook.

2. Architecture

The architecture of RucioUI is a standard web stack, using off-the-shelf software. A DNS loadbalanced Apache is used as the web-server, with *mod_wsgi* as the Python container. Each view in the RucioUI is implemented as a stateless and standalone *web.py* application, bound to a separate *mod_wsgi* thread. This allows to horizontally scale the RucioUI, should it ever be needed. The frontend itself is written using the Foundation CSS framework [5], and data is loaded via the Rucio REST API asynchronously using jQuery [6]. Data is displayed either through Foundation CSS display boxes, vector-graphics based Highcharts [7], or sortable DataTables [8].

The RucioUI is fully integrated in the Rucio authentication scheme, using an x509 certificate layer atop a standard token-based approach. The tokens themselves are stored in short-lived cookies. That way, users can interact securely with the RucioUI, and automatically have access to the respective data their Rucio account is allowed to view or modify. Cross-origin resource sharing (CORS) is used heavily to protect the data between the Rucio REST API and the web browser of the user. Invalidation of tokens is handled Rucio server-side, and a custom jQuery function automatically requests new tokens when necessary.

There is no dedicated backend of the RucioUI, because it is stateless. All interaction is handled via the Rucio REST backend servers and the cookies in the webbrowser. The only backend that is partially exposed is the Hadoop Distributed Filesystem (HDFS) [9], which is used to stream very large periodic snapshots of Rucio catalogue data to the user.

3. Occupancy and accounting

The principal views of Rucio catalogue data includes datasets, files, subscriptions, and rules. Files are contained in datasets, and rules describe how datasets are distributed globally on storage systems. Subscriptions automatically create rules for newly created datasets. These data states are usually direct representations of the current transactional state of the system. However, the more important view is the aggregation over all data states, and especially their physical representation on storage systems. The occupancy and accounting views give this

Figure 2. Replication: Select datasets

Figure 3. Replication: Select RSE

information. Figure 1 gives an example of one of the occupancy views, where users can customise the aggregation in real-time. In this case, the user selected all Rucio Storage Elements (RSE) that belong to the Standard Model physics group. This will automatically evaluate all singular RSEs and aggregate the given occupancy in Highcharts. The actual data is continuously precomputed on HDFS, therefore the view is very responsive and does not require a full scan of the Rucio database every time a user requires a custom aggregation.

4. Requesting replication rules

This view is a convenient way for the user to request one or more rules with a simple interface. The view is split in three sequential parts: In the first part as shown in Figure 2 the user can search and select for datasets that they want to be replicated. Therefore they can either provide a dataset name or a wildcard. If a wildcard is used a table with all matching datasets is shown where the user can select the ones they want to be replicated. There are two accepted ways of searching for a did. The first possible format is **scope:name**, where **name** can be either the full name, a partial name with a wildcard or just a wildcard. The second format is just **name** if the scope can be automatically extracted from the name. In the example in the figure the second option can be seen. The search is for **data15_cos.***, that is, everything in the scope **data15_cos** that starts with the pattern **data15_cos.** will be displayed.

In the next step, the user can select the RSEs to which they want to replicate the data. To do this they can again either provide a full RSE name or an RSE expression. If it is an RSE expression all the matching RSEs will be displayed and the system will automatically pick a suitable RSE from this set. In the example in Figure 3 the RSE expression **tier=2&cloud=DE&spacetoken=ATLASLOCALGROUPDISK** is used, which will result in a rule that will replicate the selected datasets to any Tier-2 LOCALGROUPDISK in the German cloud.

In the last step, as shown in Figure 4, the user can select some options for the rule. These include the type of grouping, the life-time of the rule, the number of copies if an RSE expression is used and an optional comment.

5. Backlog monitoring

The backlog monitoring, as shown in Figure 5, gives an overview of all rules for a certain account and activity together with the name of the rule, endpoint, status, creation date, data type, project, stream, version as well as the number of locks in state OK, REPLICATING and STUCK. Locks are the constituent elements of rules and describe the actual physical location of files. The view is filterable by all of these fields as well as the age of the rules. Detailed

Account	Activity	Load			Get Link					
Rules	IO Type									
Rules										
Endpoint	State	Date	Project	Stream	Age					
CERN-PROD-DEIVED	ESD	ESD	cesr15_cox		4 hours					
Search:										
Name	Endpoint	State	Creation Date	IO Type	Project	Stream	Version	OK	Replica	Block
cesr15_cox000003.physica_Medibox	CERN-PROD-DEIVED	Transfer	Pl, 24 Apr 2015 09:44:47 UTC	ESD	cesr15_cox	physica_Medibox	0	933	7	0
cesr15_cox000003.physica_ICCoxr	CERN-PROD-DEIVED	Transfer	Pl, 24 Apr 2015 09:16:04 UTC	ESD	cesr15_cox	physica_ICCoxr	0	921	9	0
cesr15_cox000003.express_aggrss	CERN-PROD-DEIVED	Done	Pl, 24 Apr 2015 09:43:19 UTC	ESD	cesr15_cox	express_aggrss	0	99	0	0
cesr15_cox000003.physica_Medibox	CERN-PROD-DEIVED	Done	Pl, 17 Apr 2015 09:43:19 UTC	ESD	cesr15_cox	physica_Medibox	0	60	0	0
cesr15_cox000003.physica_Coxr	CERN-PROD-DEIVED	Done	Pl, 17 Apr 2015 09:43:19 UTC	ESD	cesr15_cox	physica_Coxr	0	60	0	0
cesr15_cox000003.physica_ICCoxr	CERN-PROD-DEIVED	Done	Pl, 17 Apr 2015 09:43:19 UTC	ESD	cesr15_cox	physica_ICCoxr	0	64	0	0
cesr15_cox000003.physica_Coxr	CERN-PROD-DEIVED	Done	Pl, 17 Apr 2015 09:43:19 UTC	ESD	cesr15_cox	physica_Coxr	0	840	0	0
cesr15_cox000003.physica_ICCoxr	CERN-PROD-DEIVED	Done	Pl, 17 Apr 2015 09:43:19 UTC	ESD	cesr15_cox	physica_ICCoxr	0	830	0	0

Figure 5. Rule backlog monitoring

information can be displayed for each individual rule and links to external services like the DDM dashboard [10] and the FTS monitoring [11] are provided. This view can help to easily spot problematic rules, for example, by filtering all rules that are older than one day and are still replicating. The current selection of fields can be saved by generating a URL that then also can be shared.

6. Periodic snapshots

Some use cases require very large volumes of data. For example, a site administrator might want to check if the files on their storage system correspond correctly to the entries in the Rucio catalogues – a consistency check. To ease the load on the Rucio database for such queries, periodic snapshots of the Rucio database are written to Hadoop HDFS. Users can then download these snapshots via the RucioUI with a simple HTTP call.

The enabling software is a custom Java Servlet, running in an Apache Tomcat container, embedded in the RucioUI, which validates the user credentials, and streams the data directly from HDFS. As the central Hadoop instance in CERN IT provides more than a Petabyte of storage, it becomes very easy to make very large snapshots easily and readily available. Furthermore, the RucioUI provides a detailed description of the snapshots, as most of them have a large number of columns with different datatypes.

7. Access monitoring

The workload, that is, API requests put on Rucio is analysed using Apache logs as input. Rucio currently uses eleven hosts as backends for serving API requests, to cope with an average load of 270 Hz. Each backend, running an Apache web server, provides the following request data:

- Timestamp
- Hostname
- Loadbalancer IP
- Client IP
- Request ID
- HTTP Status
- Request size
- Response size
- Response time
- API Call, i.e., first line of request
- Security Token
- User Agent
- Client Script

The data collected this way is streamed to an HDFS cluster using Apache Flume [12] with an hourly log rotate. This way, the statistics presented in the web application are at most one hour behind. As an additional security mechanism, and to keep the number of files on HDFS low, each day at midnight the hourly rotate files are replaced by one daily rotated file coming

directly from each host system. This redundancy has proven itself already very useful during periods with HDFS or network hiccups, where data loss was successfully averted.

Most of the logged data is pretty standard, that is, Apache combined log format, but a few are custom header fields or their usage is not obvious, therefore a short explanation for each field is given in the following:

- **Request ID:** Apache's *unique_id_module* is used to create a unique ID for each request. By passing on this ID through the API, one can keep track of internal method calls triggered by a certain request.
- **Request Size:** Some of Rucio's API calls are POSTs with plenty of data in their body. Persisting all data of each POST request would not only saturate the host system IO capacities, it would also result in a massive amount of data. Thus we decided to keep only the size of the request as this already allows to get an estimation about the characteristics of this call which is sufficient in most situations.
- **User Agent:** If Rucio client software is used, that is, the Python client libraries, the used version is indicated in this string e.g. *dq2-clients/0.3.0* or *rucio-clients/0.3.1*. But also clients or scripts developed by users are logged e.g. *Java/1.6.0_45*, *curl/7.40.0*, *ARC*, ... Using this information, Rucio developers can keep track how the performance of API calls evolves from release to release. Further can Rucio operators see which client has been used in case of unexpected behaviour or compatibility issues.
- **Client Script:** The client script ID is the name of the script which initiated the API call combined with its first positional argument, for example when using the official clients *dq2-list-dataset*, *rucio::list-dids*. But also home cooked scripts from users are tracked this way e.g. *sonar_test_monthly.py* or *we_want_information.py*. This information is used to correlate the workload put on the API with the executed CLI script. For example the API calls originating in a call of command *dq2-list* could be significantly reduced with this information. It further enables to gain deeper understanding of user workflows and support them when optimising them. The latter is especially useful during the ongoing migration from the DQ2 API to the Rucio API.

8. Post-Processing of Log Data

The data collected as described above aggregates to roughly 17GB per day. This is too much for a browser session, and not only needs to be presented inside a web application but also to be understood in one view. Therefore we post-process them every time the log files are rotated i.e. every hour/every day to extract only specific correlations included in the raw data.

As data is already stored on HDFS, PIG scripts are used to execute Map-Reduce jobs for data aggregation [13]. Because only the totals per day are of interest, data is aggregated from the granularity of microseconds, as reported by Apache, to days. Therefore we count the hits or requests and calculate the sum of bandwidth or response size and response time per unique entry. Currently we generate five distinct aggregated views:

- **Per Account** requires the *account name* from the security token to be unique.
- **Per API Class** maps the *requested resource*, that is, API call against API classes for which the sums are created.
- **Per Country** takes the *client's IP* addresses and assigns it to a country based on the GeoLite2 database.
- **Per Resource** is grouped per *URI* e.g. *POST /replicas/list*
- **Account Details** shows the resources accessed per account, and is a combination of *per account* and *per resource*.

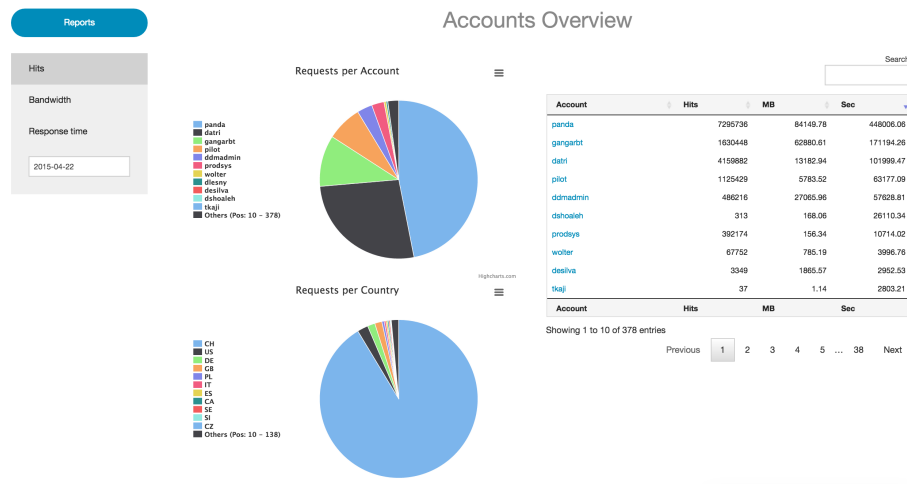


Figure 6. Rucio WebStats: Comparison between accounts

Doing so allows to get the desired information in files with a couple of megabytes up to 300MB for the largest per days. Files which are few megabytes in size are transferred directly to the browser and filtered/aggregated on the client using Javascript. Larger files support server-side filtering to keep loading time reasonable. This is implemented using a Tomcat server which reads/filters the data from HDFS and responds it as *text/csv typed data*.

For example, if all resources accessed by a given account are requested, the file, originally around 300MB, is filtered at the server to only include data related to the given account. Furthermore the user can provide the number of results that should be returned. For example, only the top 1000 matches should be returned. Using this sort of threshold further decreases response times and server load as not the complete file needs to be parsed in order to serve a request. In practice, for Rucio UI it was noticed that every request with more than 20.000 lines in response starts to take an inconvenient long time to be responded, and significantly slows down or even kills an average web browser session.

9. The Web Front End (Rucio WebStat)

The web application is divided into several *Reports*. Each report is focused on one type of resource, for example, webstats/accounts. They start by showing an overview of how the entities of resource compare to each other. For example, in Figure 6 it is shown how accounts compare to each other. Each page further includes a *date picker* and a selector for *hits*, *bandwidth*, and *response time*, allowing to select what data should be visualised.

To investigate a resource type in more details, e.g. one specific account, the details page of the resource can be requested, for example, webstats/accounts/panda. On this page, multiple detailed plots are shown explaining the accounts activity. For example, which API classes are called with which client by a given account, as shown in Figure 7 or which user agents are used by this account, as shown in Figure 8.

Each detail view further includes a data table including the data used for the plots. In this table a user can easily search and find links to additional information, for example, links to check the state of requested rule or dataset. The same construct as described for accounts, is also provided for API classes, scripts, and resources. It should further be noted that the web application follows a REST inspired approach, which allows users to easily bookmark the entities they investigate frequently.

The implementation of the UI is based on the ZURB Foundation framework [5] to get

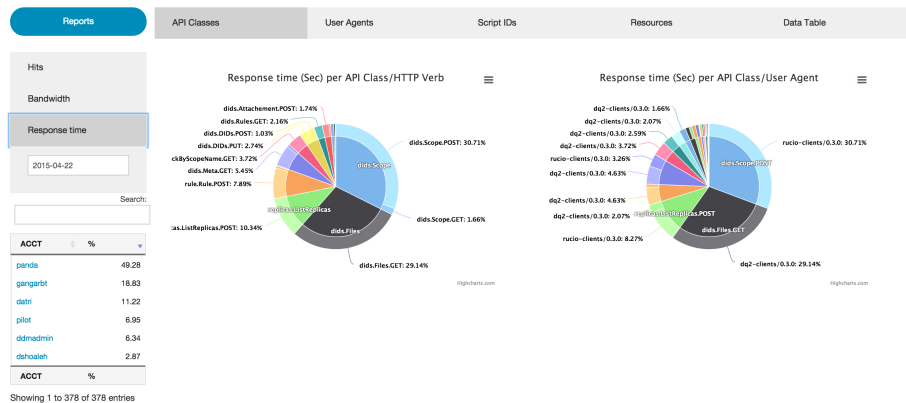


Figure 7. Rucio WebStats: API Classes called by Account

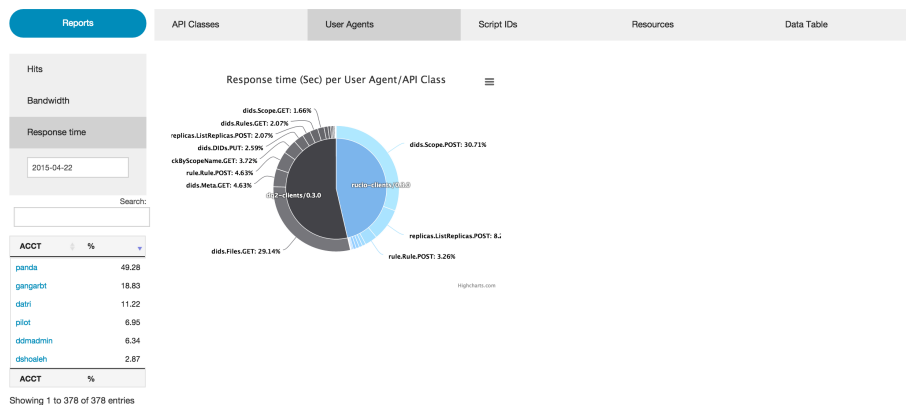


Figure 8. Rucio WebStats: User Agents used by Account

responsive behaviour. Plots are created by using Highcharts and tables utilise the DataTable plugin for jQuery. All data for Highcharts and DataTables are filled asynchronously.

10. Performance measurement

Performance metrics, coming from several services and hosts, are reported against Graphite [14] with a receiving StatsD server [15] in front. Combining Graphite and StatsD allows for high frequency real-time data taking of numeric time series. StatsD is a light-weight UDP server, aggregating received data into distinct metrics and calculating basic statistics like lower, upper, mean, or rate automatically. Eventually the data is periodically flushed to Graphite which makes them persistent in RRD databases files [16] for later analysis.

In Rucio, as well as in StatsD, three types of metrics are supported:

- (i) **Counters** are used to count sums per time period. Whenever a new value is reported to a metric of type counter, it is added to its current value and assigned to the *sum* appendix of the metric. StatsD also maintains a further appendix named *rate* which provides the sum in the form of events per seconds. For example, if the flush interval is 60 seconds, and the sum of counter adds up to 90, the rate is 1.5, independent of the number of reports.
- (ii) **Gauge values** are used for metrics taken less often than the defined flush interval. For counters, if a metric was not reported since the last flush, it would be reported with *null*

at the next flush. This can be rather inconvenient when analysing or plotting data, as null values interfere with derivation calculations, or just ruin the plot by making it extremely spikey. To avoid these situations, StatsD will continue reporting the last known value of a gauge metric until it receives a new one. This allows more flexibility in terms of flush intervals and metric reporting.

- (iii) **Timers** are used to monitor how long it took to execute a certain operation. StatsD aggregates the reported execution times as an average to the metric, and preserves the highest (*upper*) and lowest (*lower*) appendices. It also keeps track of how many reports were received in the flush interval (*count*) and how many Hz (*count_ps*) this represents. Having all this information allows to derive fine grained understanding about how expensive specific pieces of code are. It is a valuable source of information when identifying performance bottlenecks and to keep track of potential performance improvements.

Graphite not only has the ability to store the data in RRD files, it also comes with a web front end to plot them. This PHP-based web application allows to combine and transform stored data in several ways. A complete list of supported methods is provided in [14]. Using the Graphite Web Composer, one can transform and combine several metrics into one plot to give a comprehensive view on the intended service or operation.

With all this fine grained information in a single place, it stands to reason to use it to have an automated observation of this data and triggering alarms if something is off. To do so, the URL API of Graphite is used to request the data in JSON format. It should be noted that all the possibilities of combining and transforming are also available here. Together with data filtering, which is also supported by Graphite, this is a very powerful foundation for complex tests. In this implementation, it can be defined how often a certain threshold is allowed to be exceeded or fallen behind, by counting the null values responded by Graphite. Depending on this number, either OK, a warning, or a critical error is reported.

11. Summary

The RucioUI serves as the principal interface to ATLAS Distributed Data Management for regular users. They can view information about the experiment data, inspect their details, request data transfers, and monitor system throughput and performance. The RucioUI is built atop a scalable stack, using distributed, load-balanced, and asynchronous frameworks. It was officially put into production in December 2014, and the user response was very favourable. Eventually, the RucioUI has superseded many of the previously custom-built monitoring pages in ATLAS Distributed Computing.

References

- [1] The ATLAS Collaboration et al 2008 JINST 3 S08003
- [2] V Garonne et al 2014 *J. Phys.: Conf. Ser.* 513 042021
- [3] I Bird, Annual Review of Nuclear and Particle Science, Vol. 61: 99-118, November 2011
- [4] R W L Jones and D Barberis 2010 *J. Phys.: Conf. Ser.* 219 072037
- [5] ZURB Foundation URL <http://foundation.zurb.com/>
- [6] jQuery - Asynchronous JavaScript Library URL <https://jquery.com/>
- [7] Highcharts URL <http://www.highcharts.com/>
- [8] DataTables URL <http://datatables.net/>
- [9] Hadoop - Reliable, scalable, distributed computing URL <https://hadoop.apache.org/>
- [10] ATLAS DDM Dashboard - Transfer, staging, deletion URL <http://dashb-atlas-ddm.cern.ch/ddm2/>
- [11] FTS3 Monitoring - File transfers URL <https://fts3.cern.ch:8449/fts3/ftsmon>
- [12] Apache Flume URL <https://flume.apache.org/>
- [13] DataTables URL <http://datatables.net/>
- [14] Graphite - Scalable Realtime Graphing URL <http://graphite.wikidot.com/start>
- [15] StatsD - Measure Anything, Measure Everything URL <https://github.com/etsy/statsd/wiki>
- [16] RRD - Round Robin Database URL <http://oss.oetiker.ch/rrdtool/>