Scalable and fail-safe deployment of the ATLAS Distributed Data Management system Rucio

M Lassnig[1], R Vigne[2], T Beermann[1], M Barisits[1], V Garonne[3], C Serfon[1], for the ATLAS Collaboration

[1] ATLAS Data Processing, Physics Department, CERN, 1211 Genève 23, Switzerland
[2] Institute for Astro- & Particle Physics, University of Innsbruck, 6020 Innsbruck, Austria
[3] Department of Physics, University of Oslo, 0316 Oslo, Norway

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

Abstract. This contribution details the deployment of Rucio, the ATLAS Distributed Data Management system. The main complication is that Rucio interacts with a wide variety of external services, and connects globally distributed data centres under different technological and administrative control, at an unprecedented data volume. It is therefore not possible to create a duplicate instance of Rucio for testing or integration. Every software upgrade or configuration change is thus potentially disruptive and requires fail-safe software and automatic error recovery. Rucio uses a three-layer scaling and mitigation strategy based on quasi-realtime monitoring. This strategy mainly employs independent stateless services, automatic failover, and service migration. The technologies used for deployment and mitigation include OpenStack, Puppet, Graphite, HAProxy and Apache. In this contribution, the interplay between these components, their deployment, software mitigation, and the monitoring strategy are discussed.

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.

Rucio is one of the underpinnings of the distributed computing stack of ATLAS. Many components interact with Rucio, most importantly the the workload management system *PanDA* [5], therefore an uninterrupted service is required. If Rucio fails to operate properly, jobs assigned for execution in PanDA will fail, because the jobs cannot find and act on their data anymore. It is therefore of utmost importance that Rucio performs efficiently and fault-tolerant. The two main objectives are thus to make Rucio scalable to the workload of ATLAS distributed computing, and fail-safe with a minimum amount of human intervention.

This paper is structured as follows. First, the infrastructure for the deployment of Rucio is described. This includes both the node and configuration management. Second, the software architecture is described, which enables automated horizontal scalability, makes the service itself fault-tolerant, and allows easy monitoring of system runtime characteristics for anomaly

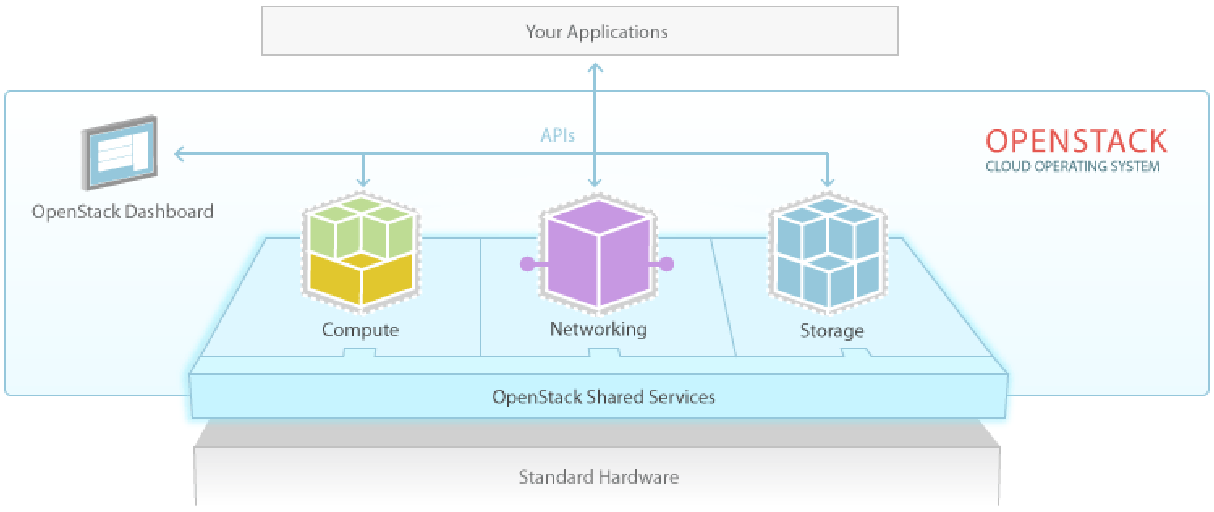

Figure 1. OpenStack [6] is the foundation of the CERN IT data centre.

mitigation. The paper concludes with a discussion on the design choices, and their perceived benefits in the first months of Rucio in production.

2. Infrastructure

The infrastructure is built upon two parts, the node management system OpenStack [6], and the configuration system Puppet [7]. Both services are hosted by the CERN IT data centre [8].

2.1. Node management

OpenStack is a cloud operating system that controls large pools of compute, storage, and networking resources throughout a data centre, all managed through a dashboard that gives administrators control while allowing users to provision resources through a web interface [6]. Figure 1 shows the basic architecture of OpenStack. *Compute* uses one of multiple supported hypervisors in a virtualized environment. The CERN IT data centre uses the Linux internal KVM. *Storage* provides a fully distributed, API-accessible storage platform that can be integrated directly into applications or used for backup, archiving and data retention. Block Storage allows block devices to be exposed and connected to compute instances for expanded storage, better performance and integration with enterprise storage platforms. *Networking* is a pluggable, scalable and API-driven system for managing networks and IP addresses. Detailed information about OpenStack can be found in their documentation [9].

2.2. Configuration management

OpenStack by itself is not useful for systems like Rucio. The nodes need to be configured appropriately, such that Rucio services can run. This is done via a configuration mechanism that is layered atop OpenStack. The two systems Puppet [7] and TheForeman [10] are used to accomplish this. The CERN IT data centre provides setups of OpenStack together with Puppet and TheForeman.

Puppet usually uses an agent/master (client/server) architecture for configuring systems, using the Puppet agent and Puppet master applications. Puppet configures systems in two main stages: (1) compile a catalogue, (2) apply the catalogue on the node. The task of the Rucio administrators is to write the manifests that comprise the catalogue, such that the manifests can be found, compiled into a catalogue, and eventually applied. This process is automated by the

```
class hg_voatlasrucio::rucio::authentication {
  class { 'hg_voatlasrucio::common::selinux': }
  class { 'hg_voatlasrucio::common::iptables': }
  class { 'hg_voatlasrucio::common::httpd': }
  class { 'hg_voatlasrucio::common::rucio': }
  class { 'hg_hadoop::default::base': }

  file {
    '/etc/httpd/conf.d/rucio.conf':
      path => '/etc/httpd/conf.d/rucio.conf',
      content => template('hg_voatlasrucio/rucio/auth.conf.erb'),
      notify => Service['httpd', 'memcached'];
  }
}
```

Listing 1. Example puppet manifest for Rucio authentication nodes

CERN IT data centre. The manifests themselves are a declarative description of the software needed in a given configuration, and follows a Ruby-like syntax.

Listing 1 gives an example how the manifest entry for Rucio authentication looks like. First, manifests are hierarchically structured into hostgroups, denoted by double colons. In this case, the hierarchy is shown with the top-level hostgroup *hg_voatlasrucio*, and then *rucio/authentication*. The first four clauses include other classes from the top-level hostgroup, however, from a different branch in the tree, called *common*. Accordingly, this is the basic node setup with SELinux, a firewall, a web-server, and the Rucio software itself. Additionally, a fifth include imports from another top-level hostgroup that was not written by the Rucio administration team. The Hadoop top-level hostgroup is provided by CERN IT, but can be included, as hostgroups are all globally available. The *file* clause describes that the given file */etc/httpd/conf.d/rucio.conf*, that is, the web-service configuration, is to be loaded from a standard Ruby *ERB* template that is stored in Puppet. To ensure that other services pick up changes to this configuration file, whenever this file is changed, a notification is registered and both the web-server and the memory-cache are restarted.

Using this declarative description of nodes is straightforward and brings many benefits, most importantly short and concise configurations. There is, however, one caveat, as catalogues are not processed linearly; this means that in-order execution is not guaranteed. This is especially troublesome when software is to be installed that requires certain preconditions. If such preconditions are not represented in the Puppet Standard Library then custom scripts have to be written to ensure proper ordering. For example, directories might be missing which can cause installation of software to fail.

Finally, the configuration also has to take care of secrets, for example, database passwords. As the Puppet catalogues are publicly readable, an administrator cannot simply put the passwords in them. This part is handed off to a separate service *Teigi*. Manifests just need to reference a key, which is automatically expanded by Teigi during compilation of the catalogue with the actual secret value. The secret value can be set via a separate commandline tool, separate from the rest of the configuration; this gives another layer of security.

3. Software

Distinct from infrastructure management, Rucio was designed to mitigate potential problems in software early. It is horizontally scalable, fault-tolerant, and allows direct component monitoring.

3.1. Horizontal scalability

In order to meet reliability and scalability requirements, two quad-core nodes, each running HAProxy [11] with four concurrent processes are used as load balancers. The nodes accept HTTPS requests from everywhere, while HTTP requests are only accepted from specific services inside CERN. In case of HTTPS requests, HAProxy is also in charge of SSL termination. Measurements have shown that early SSL termination reduces roughly 25% of CPU load on the backend nodes.

To provide stable service levels, and optimise the setup of the nodes for specific purposes, HAProxy defines five different backends with distinct sets of nodes. A set of access control lists (ACLs) defines which backend will receive a request. For example, requests for resources related to *traces*, that is, URIs which start with */traces/...*) will go to a set of nodes optimised for this specific type of requests. Another distinction is made depending on the HTTP method. HTTP GET requests will go to a set of nodes optimised for reading data from the database, while HTTP POST requests will go to nodes optimised for writing into the database. Last, the account issuing the request is considered. In order to guarantee stable conditions for production and analysis systems, dedicated sets of nodes are provided. Inside each backend, HAProxy balances the load using the *least connection* approach. Eventually, two HTTP header fields are added, that is, one including the original source IP named *X-Forwarded-For*, and one indicating which of the host load balancer hosts forwarded the request, named *X-Requested-Host*.

In addition to serving user requests, Rucio also has services which run continuously. This includes services like the *conveyor*, which is responsible for scheduling and managing file transfers on the WLCG, or the *reaper*, which is responsible for file deletion. These services are horizontally scalable as well. Adding a new conveyor or reaper node via OpenStack/Puppet will scale-out the service as required. This is enabled through two mechanisms that complement each other, heartbeats and avoidance. Every service periodically sends a heartbeat to the central database, specifying a tuple *(executable, hostname, pid, thread_id, thread_name, last_modified)*. This gives an instant view on the currently running services. Potentially stuck services can be spotted easily by older *last_modified* timestamps. The return value of the heartbeat is the aggregate number of threads for a given *(executable, hostname, pid, thread_id, thread_name)*, and the assignment of an integer id. This integer-id is then used by the avoidance algorithm in each service to ensure that no other threads work on the same entries in the queue. For example, this avoids that two conveyors try to transfer the same file each. Through dynamic allocation of these integer ids, it is possible that new services can be brought up and all existing services will automatically adapt their share of the work.

3.2. Fault-tolerance

Each HAProxy process executes periodic health checks for each backend host by issuing a Rucio ping request. This verifies that the nodes are still alive, and Rucio was started inside Apache and can respond appropriately. If a node fails to respond, it will be excluded from the load balancing and reported as being 'DOWN'. HAProxy will continue checking the node and include it in the load balancing as soon as it successfully responds to a Rucio ping request.

The fault-tolerance of Apache containers for the Rucio backend are straightforward. Their configuration forces a container restart after a given number of failed requests. Though a somewhat drastic approach, this has effectively prevented known memory leaks from Apache itself.

The fault-tolerance of Rucio code is built atop the statelessness of requests and a stringent database layer, using SQLAlchemy [12]. Due to statelessness, a single hanging request cannot block others. In case of a code crash, the database connections within an Apache container must be reestablished, to continue serving new requests while ensuring transactional safety. This is ensured through Python annotations on all requests. A negligible amount of CPU is thus used on every request to check for availability of connections in the connection pool, and missing connections are reestablished. The major complication in these annotations is the inconsistent error handling across multiple database backends, which had to be implemented specifically for SQLite, MySQL, PostgreSQL, and Oracle.

3.3. System measurements

Performance metrics, coming from several services and hosts, are reported against Graphite [13] with a receiving StatsD server [14] in front. Combining Graphite and StatsD allows 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 [15] 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 counters adds up to 90, the rate is 1.5, independent of how many separate reports were sent.

(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*). It also keeps track of how many reports where 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 pieces of code are. It is a valuable source of information when identifying performance bottlenecks and to keep track of potential performance improvements.

In Rucio four methods are provided to report these metrics. Examples of how this methods can be used are shown in Listing 2.

Three of them are implemented as a single statement i.e. *record_counter* (line 2), *record_gauge* (line 5), and *record_timer* (line 8). Each of these methods accepts the name of the metric and the value to be reported. The fourth method, *record_timer_block* (line 11) is implemented as a block statement, which times the execution of the provided code block automatically and reports it when execution has finished. As Rucio has a considerable number of *bulk methods*, these statements further support provision of numbers (e.g. the number of files included in the bulk submit) which are used to normalise the reported execution time (line 13).

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 combination and transformation of stored data in several ways. A complete list of supported methods is provided in [13]. 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. For example, Figure 2 shows the number of concurrent sessions per backend reported by HAProxy.

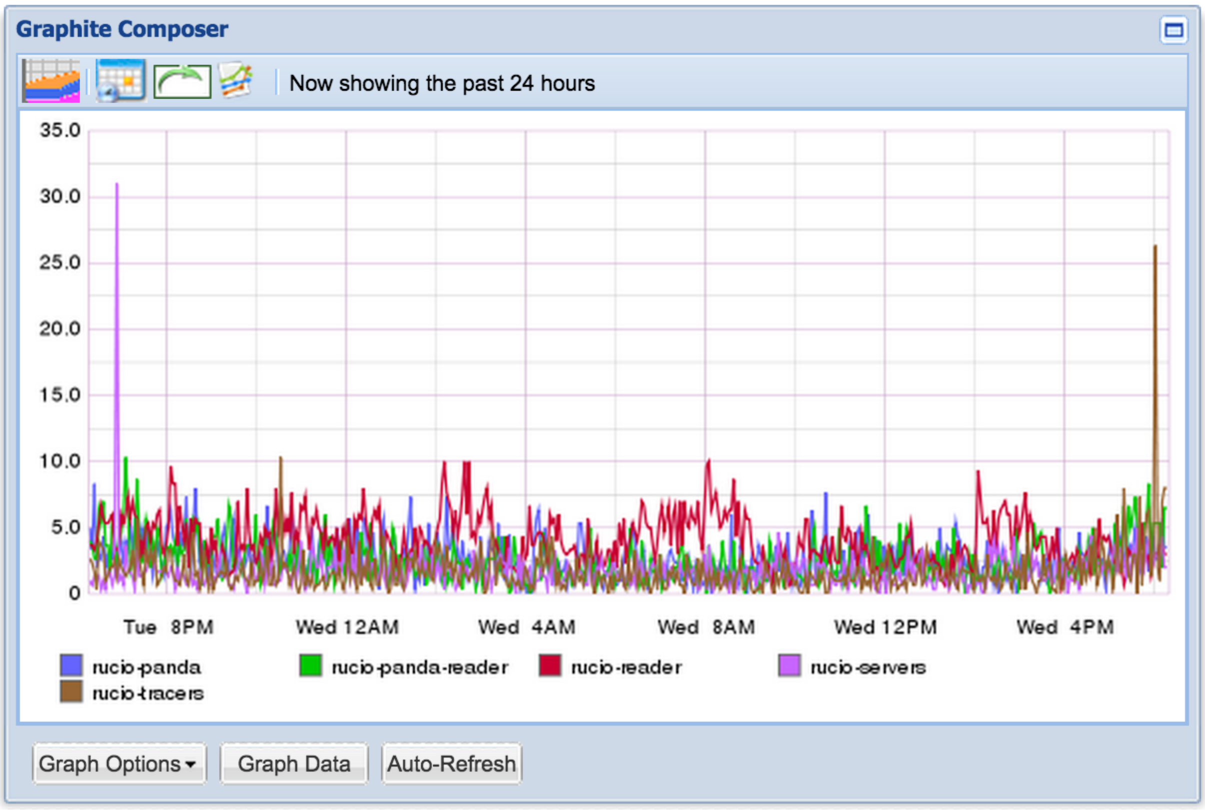

Figure 2. Graphite Composer: Number of concurrent sessions per backend

```
# Increase test.counter by ten
monitor.record_counter('test.counter', 10)

# Set test.gauge to new value ten
monitor.record_gauge('test.gauge', 10)

# Report test.runtime with 500 milliseconds
monitor.record_timer('test.runtime', 500)

# Time execution of code block
monitor.record_timer_block(['test.timer', ('test.timer_normal10', 10)]):
```

Listing 2. Examples of Rucio monitoring methods

Usually, information like this can only be interpreted properly if shown in combination with other information. For example, if the number of concurrent sessions is reasonable or not can only be judged if compared with the requests rate at the time and/or the average response time. To support users to get a comprehensive view of such correlating data, Graphite supports the creation of *dashboards*. Dashboards are an ordered set of plots showing the same period of time. Figure 3 shows an example for a dashboard providing a comprehensive view about the current requests. It shows the request rate together with the load per backend, and also includes information of each node within a backend. It can only be judged if everything is in order when

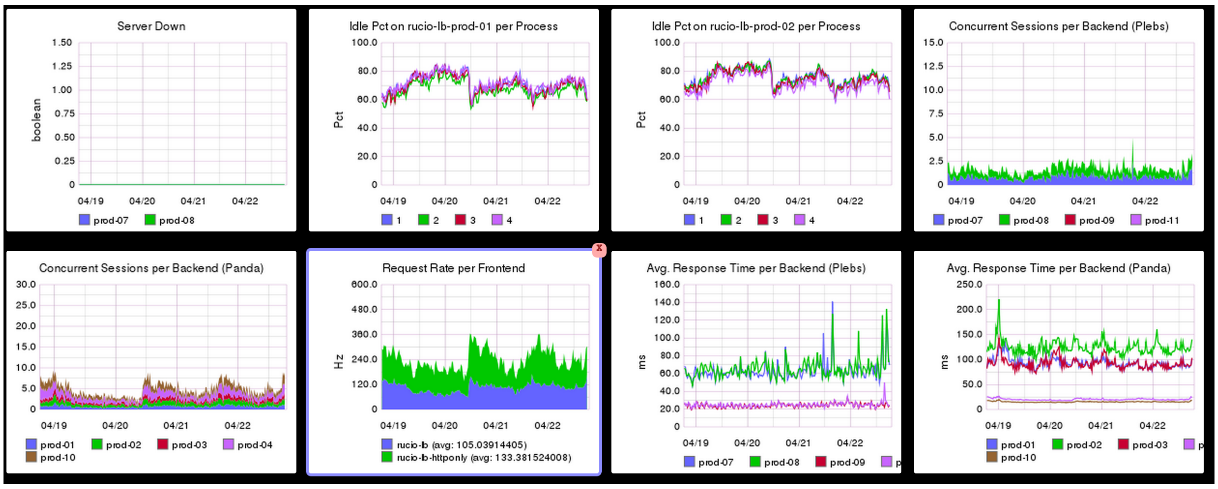

Figure 3. Graphite Dashboard: Overview of the evolution of request distribution by HAProxy

```
url = 'http://%s/render?format=json&from=-%smin&target=%s' %
       (GRAPHITE_URL, int(options.period)+1, options.target)
r = requests.get(url).json()

exit_code = OK
for target in r:
    for db in target['datapoints']:
        if not db[0]:
            counter +=1
        if (counter > options.warning):
            exit_code = WARNING
        if (counter > options.crtitical):
            exit_code = CRITICAL
sys.exit(exit_code)
```

Listing 3. Simplified example of a Nagios probe using Graphite data

one has all this information at hand.

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 Listing 3 we provide an example of how a Nagios [16] probe, utilising this data, can be implemented.

In this implementation, it can be defined how often values are allowed to fall below or above a certain threshold, by counting the null values returned by Graphite. Depending on this number, either OK, a warning, or a critical error is reported. In Listing 4 an example how this probe is used to check the availability of the load balancer is shown.

The target represents the HAProxy nodes, and reports the idle percentage of the last 30 minutes. All values above 25% are filtered. If the number of data points exceeds either 20 or 25, Warning or Critical is reported accordingly. Already in this simplified example, it can be

```
graphite2nagios
    --target
        aliasByNode(
        removeAboveValue(
            stats.rucio.monitoring.loadbalancer.*.*.Idle_pct,25),4)
    --critical 25
    --warning 20
    --from 30
```

Listing 4. How to call the Graphite Nagios probe

seen which levels of complexity can be implemented into probes with a few lines of code.

4. Summary

The OpenStack virtualised infrastructure, provided by the CERN IT data centre, has proven to be stable and reliable. In cooperation with the Puppet configuration management system, a comprehensive suite for installation, configuration, and maintenance is available. Rucio leverages all features for load-balancing and fault-tolerance provided by OpenStack and Puppet, and implements custom handling on the application side as well. In total, this gives three parachutes for Rucio, which also serve as scalability features: the automatic redistribution of requests in case of Rucio node failure, the automatic reconfiguration in case of external system failure, and the automatic restart of services in case of any unexpected problems. This is coupled with a tightly integrated monitoring system, that allows quasi real-time monitoring of system performance characteristics, and direct notification of persons on-call in case of alarms. Especially the real-time monitoring has proven to be a useful tool for ATLAS data management operations, because it allows cascading events from external systems to be followed.

The most important future work will include the networking component of OpenStack. This is not yet used by CERN IT, but will enable the provisioning of dedicated network links, which can help improve latency for catalogue search and data transfer. This activity will complement the idea of virtual circuits, that is, software-defined networks, that is proposed within the WLCG.

References

[1] The ATLAS Collaboration 2008 JINST 3 S08003
[2] V Garonne et al on behalf of the ATLAS Collaboration 2014 J. Phys.: Conf. Ser. 513 042021
[3] I Bird 2011 Annual Review of Nuclear and Particle Science 61 99-118
[4] R W L Jones and D Barberis 2010 J. Phys.: Conf. Ser. 219 072037
[5] T Maeno et al on behalf of the ATLAS Collaboration 2014 J. Phys.: Conf. Ser. 513 032062
[6] OpenStack – Open Source Cloud Computing Software URL `https://www.openstack.org/`
[7] Puppet – IT Automation for System Administrators URL `https://puppetlabs.com/`
[8] Ramon Medrano Llamas et al 2014 J. Phys.: Conf. Ser. 513 032066
[9] OpenStack Documentation URL `http://docs.openstack.org/`
[10] Foreman – Lifecycle management for physical and virtual servers, url = http://theforeman.org/
[11] HAProxy – The Reliable, High Performance TCP/HTTP Load Balancer URL `http://www.haproxy.org/`
[12] SQLAlchemy – The Python SQL Toolkit and Object Relational Mapper URL `http://www.sqlalchemy.org/`
[13] Graphite – Scalable Realtime Graphing URL `http://graphite.wikidot.com/start`
[14] StatsD – Measure Anything, Measure Everything URL `https://github.com/etsy/statsd/wiki`
[15] RRD – Round Robin Database URL `http://oss.oetiker.ch/rrdtool/`
[16] Nagios – The Industry Standard in IT Infrastructure Monitoring URL `http://www.nagios.org/`