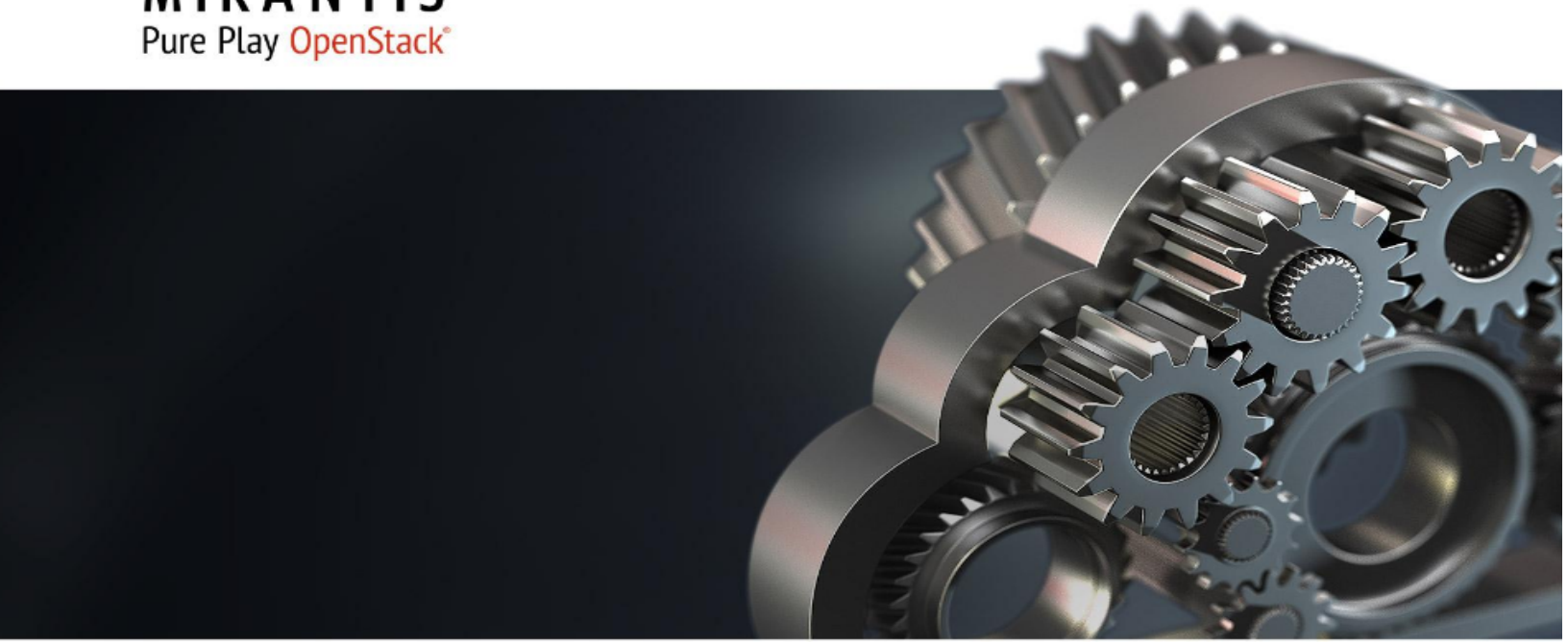
MIRANTIS
Pure Play OpenStack®



# OPENSTACK

## PERFORMANCE AND SCALE TESTING METHODOLOGY

DR. GEORGY OKROKVERTSKHOV, DINA BELOVA

## Introduction

Cloud computing is a complex and evolving process. Its definition can be as simple as

> a style of computing in which scalable and elastic IT-enabled capabilities are delivered as a service using Internet technologies[15]

, at the same time it may require a separate article to explain what cloud computing is. The main point is that there are several characteristics which are mandatory for the system to be called a cloud: scale, elasticity, and as a Service. These three characteristics are areas of interest for performance testing and scalability testing. Clouds, being complex systems, require special knowledge about their behavior to be able to assess the capacity, effectiveness, usability and operability of the cloud, not only from the base resource availability standpoint, but also from the ability to manage and deliver these resources to the end user. In this document the team of performance and scale engineers from Mirantis. Inc provides a summary of their knowledge and experience gained during several years of working, operating, and testing OpenStack clouds of different scale.


# 1. Measurements

## 1.1 Measurement Types

There are several types of measurements which are common in performance and scalability testing. A single value measurement is often used for finding a specific characteristic of some service or process. Examples of such measurements include bandwidth of a network connection, maximum density of VMs on the physical compute nodes, maximum number of objects in an object storage system, and maximum size of an object that can be stored in the object storage system. Another type of measurement is a time series, which gives an understanding of the processes which occur in a complex distributed system. Time series are useful for understanding the transition process when a system is changing between two different states. For example, HA and DR situations are specifically interesting as they are by definition transition processes. Both react in the event of failure, which changes the system state. Time series are the source of information for capacity and scale formulas where a specific characteristic with respect to some parameter is represented as a mathematical model in form of an equation or a rule. These formulas are the key results of scale and performance testing as they can be used by architects and cloud operators to predict the capacity and effectiveness characteristics of any particular design for a cloud.

## 1.2 Measurement Process

Measurement process plays an important role in testing overall, and in performance and scale testing in particular. In performance and scale testing one has to deal first with

the complexity of the system, as it is almost impossible to simplify the Device Under Test (DUT) setup. Then the complexity of the testing, which usually use load generators, fast gathering of metrics on large scales, etc., must be understood so that the precision and trueness of the measurements are not adversely affected. The importance of the measurement method is recognized by the ISO (International Organization for Standards) in their special standard ISO 5725-1 "Accuracy (trueness and precision) of measurement methods and results".

Without going too deep into the details of ISO 5725-1, the two fundamental terms of that report are briefly discussed in this section.

The first term is "precision" which defines or describes the accuracy of a measurement method[13]. Precision of the measurement method defines the susceptibility of the measurement to an influence of random errors, which can be also divided into avoidable and unavoidable errors. There are many different factors which may influence the results variability of a measurement; some of them are:

- the operator
- the equipment used
- the calibration of the equipment
- the environment and environment variations
- the time elapsed between measurements

The variability of the measurements directly influences two important characteristics of a test: repeatability and reproducibility. These reflect the minimum and maximum of variability, respectively.

The second term is "trueness" which describes the closeness of agreement between the mean value of test results and the true or accepted reference value. Trueness is a key characteristic of the measurement method as nobody is interested in wrong results provided by a wrong method. Trueness can be controlled by using reference or etalon values on etalon systems. In some measurements it is not always possible to know the true value or it cannot be measured directly. In this situation the correct selection of indirect measurements as well as an understanding of the relation between the value we want to measure and the directly measurable values in the system are of major importance.

There are known factors which might affect the trueness of the measurements:

- uncertain state of the system
- incorrect mathematical models or assumptions in direct or indirect measurements
- dependency between the measurement process and the system state when the measurement process might change the system state or trigger underlying processes which influence the measured value

Uncertain state of the system is a typical source of abnormal errors in measurement. This can be referred to as a problem of the initial state in the test. This is important for test repeatability which requires the system to behave identically for repeated measurements with the same initial conditions. Caches and network adapter devices are good examples of sources of initial condition problems. If measuring cache performance is not one of the goals of the test, the cache should be disabled so that it is possible to measure

characteristics of the system behind the cache. If it is not possible to disable caching (for example in storage testing it is difficult to disable some caches), the measurement process should be specially designed to reduce the effect of cache by using properly designed payloads or by using special statistical methods to distinguish cache effects from characteristics of the underlying system. The statistical methods for such cases are discussed further in this chapter in section 1.3.

In some cases it is possible to use the "warm-up" approach where the system is loaded with some payload and stabilized. Then this stable system state can be assumed to be in a reproducible initial state for the test. At the same time, it is important to understand that all assumptions in the measurement method should be checked on the actual system and verified for correctness.

**Note**

It is dangerous to rely blindly on the warm-up procedure as the warm-up payload can be incorrectly selected and may be inappropriate for the specific measurement type.

In our cache example, a warm-up with the same single payload may not be adequate while a random payload may lead to a proper cache state.

Incorrect mathematical models or unjustified assumptions about system behavior are other common sources of error and failure to achieve test repeatability. Wrong assumptions can lead to serious issues with the measurement method and, as a consequence, will lead to wrong or unreliable results. One of the problems here is that these assumptions are usually used not during the measurement method design but in the process of measurement process optimizations when the operator is trying to optimize the time and materials for the test. For example, the wrong assumption that two parallel requests to the API layer, which might be deployed as a server farm behind a load balancer, will be completely independent. That is, instead of a row of sequential measurements it will be possible to do them in parallel. In the case when all these API servers are using the same DB layer and the request uses the same table or table row, the parallel requests to the API will have to synchronize due to the DB layer lock. This locking can cause significant difference between the values measured via sequential single requests and the values obtained via parallel request execution. In fact, these two measurement approaches will measure two different characteristics of the service.

Dependency between the measurement process and the system state when the measurement process might change the system state or trigger underlying processes which influence the measured value is a difficult problem to deal with. This situation is quite common in complex application deployments and the measurement process should be at least aware of it and preferably be able to reduce the effects of such dependencies to a minimum. To understand the complexity of this situation let's consider the following scenario. We have a system which exposes a standard CRUD[1] REST API that uses a DB as a backend. The latency of the "Create" operation is measured by designing a test to collect statistics over 30 repeated "Create" operations. In order to bring the system to

[1] CRUD - Create, Read, Update, Delete

the initial state and exclude the possible effect of the number of objects created, each "Create" operation is accompanied by the corresponding "Delete" operation for the created object. Having the results of the 30 iterations of the "Create"-"Delete" pairs of operations we assume that we are in the situation where all errors are due to random effects and we can use a standard statistical approach to find the mean value as our measured value and the standard deviation to find the confidence interval. Unfortunately, the actual implementation of the "Delete" operation is different from what was assumed. Instead of removing the object from the database it is just marked as "deleted", so it still exists and consumes DB table space and resources. Each time a new object is created by the "Create" operation, it changes the state of underlying DB and thus each "Create" operation takes more time than the previous one. As a result, the time series of measured latencies has an increasing trend, and the measured mean value depends on the number of iterations performed in the test. To prevent such situations and to verify the correctness of the measurement process there are special statistical methods that should be used.

### 1.2.1 OpenStack Services Monitoring

It is important to pay attention to not only the control plane (CRUD) and data plane operations during testing, but also to store hardware measurements like the RAM and CPU utilization of various services. This can help to identify possible bottlenecks in OpenStack services and to improve their usability, performance, and scalability.

We use the LMA (Logging, Monitoring, Alerting) toolchain developed by Mirantis for environment monitoring and degradation research. Each service or group of services observed (MySQL, RabbitMQ, Keystone, etc.) has a different set of metrics to monitor which can effectively point to the possible performance and scalability limitations of that service.

We use the following measurements to monitor cluster state and services usability:

- virtual compute resources - number of used and free VCPUs, amount of used and free RAM and disk capacity
- cloud controllers characteristics - amount of used and free CPU, RAM, disk space
- messaging bus (RabbitMQ) characteristics - number of consumers, queues, connections and exchanges. Memory used by all queues and outstanding messages are monitored as well
- database layer (MySQL) characteristics - number and ratio of SQL commands (COMMIT, DELETE, INSERT, SELECT, ROLLBACK, UPDATE), threads used, Rx/Tx ratio and locks observed
- number of requests, connections and bytes transmitted through the Apache HTTP Server used by various OpenStack services
- HAProxy characteristics for the managed OpenStack services - frontend sessions, response rates, frontend network throughput and backend retries count, etc.
- Memcached (as a Keystone backend) characteristics - free and used cache, get/set rates, connections and network activity statistics, etc.
- Cinder, Keystone, Glance, Heat, Neutron, Nova resource details (e.g. number of

healthy services running, used and available resources, number of VMs, images, snapshots, etc., present in the cloud and more)

Knowledge of the described data allows catching OpenStack resources hung in an unhealthy state, hardware resources (CPU/RAM) usage effectiveness and leakage, OpenStack services health (which is often useful in identifying the maximum possible load a cloud can support).

## 1.3 Measurement Analysis

The measurement process produces raw data of the measured value or values. It is important to understand that even in the most careful measurements there are several sources of errors which affect the precision of the measurement and might influence the final result of the data processing.

Error sources can be roughly divided into two categories:

- Random errors due to measurement equipment error, random deviations in the measurement process, random change of the surrounding environment, or unexpected changes in the system under test
- Internal complexity of the system which has definite but complex internal processes or process sequences which are hard to predict and take into account in the measurement process

Random influence on the measurement results is a very well known effect and can be properly handled by performing statistical analyses of the results. Statistical analysis assumes that the measurement process can be repeated without change in the system or with slight changes which will not affect the measured values. Statistics are gathered during a series of repeated measurements of the measured value. The obtained series of measured data is then processed through several statistical tests. It is important to mention that the series of data can be obtained in two ways – by measuring the value on several identical systems or by performing repeated measurements over time on a single system under test. While the first approach is out of scope of cloud testing as typically there is only one cloud available, it is important to recognize that results might differ as the system might not be ergodic, i.e. an average over time is different from an average over system states.

In our team we use the following process for statistical analysis of experimental data.

### 1.3.1 Process Stationarity

The first step is to check that the measured value is a result of a stationary process, which in the case of weak stationarity means that the first moment *mean value* does not change over time, i.e. there are no value trends. For example, the keystone get_token operation response time is definitely a stationary process (Figure 1.1) while the keystone list_users operation is not (Figure 1.2).

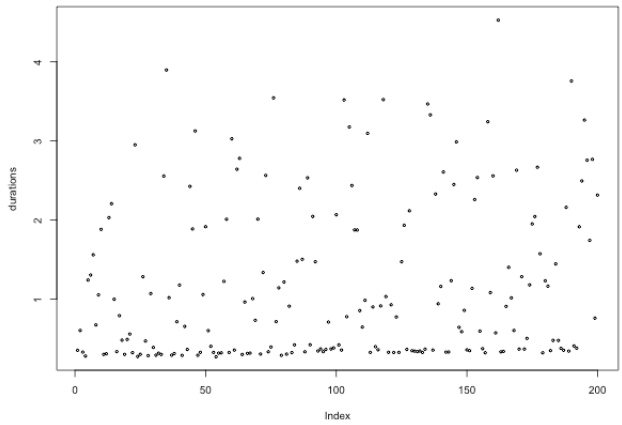



Figure 1.1: Keystone get_token operation response time deviation chart

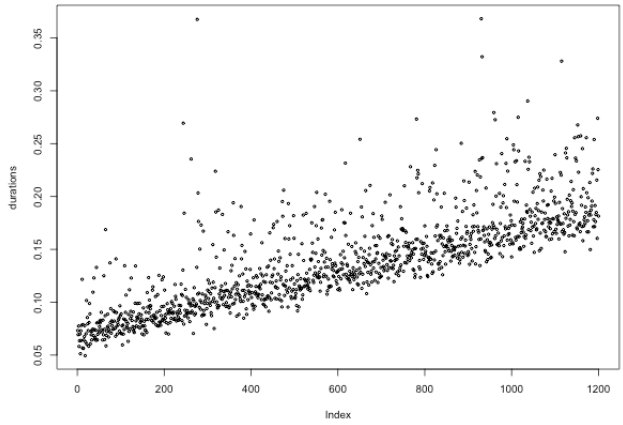



Figure 1.2: Keystone list_users operation response time deviation chart

**Note**

Stationarity verification should be the first verification of the gathered measurement data. Most statistical methods assume or require a stationary process. Before conducting further statistical analyses, the gathered data should be either stationary or transformed to stationary.

Non-stationary processes can display various types of behavior, the most common of which are trends and random walks. Trends are an important part of the information about the system as they can reveal some of the deterministic relationships between various parameters of the system and its configuration. Understanding the root cause of a trend can expose weak parts of the system architecture. For example, a growing trend in the latency of the create user operation might indicate suboptimal usage of the database layer which can create scale issues when the number of users becomes large. Almost all trending behaviors of the system should be properly documented as they usually have

direct impact on the system capacity. Deterministic relationships between parameters, revealed by trend analyses, can be presented in the form of graphs or mathematical formulas. Both can be used later by architects to predict the system behavior and capacity at different loads. The actual methods used for stationarity analysis are described in section 1.4.1.

### 1.3.2 Normal Distribution Hypothesis

The second statistical test is to verify the hypothesis that a measured value is normally distributed. If the measured value is normally distributed we can assume that all differences of measured values from their true values are the result of the cumulative influence of independent random events including measurement errors, instrument errors, and other factors. When a measured value has a normal distribution it is possible to give more specific meaning to different statistical characteristics. For example in a normal distribution, most of the values will be close to the mean value and very likely the mean value is a good approximation of the true value of the measured characteristic. It is also possible to say that 95% of all values will be within a $2\sigma$ range of the mean value as shown in Figure 1.3, where $\sigma$ is the standard deviation. This $2\sigma$ value is often used to construct a confidence interval of the measured value.

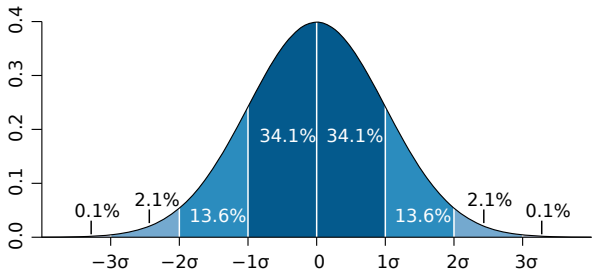



Figure 1.3: Standard normal probability density distribution and percentiles [24]

The Heat create stack operation response time is a great example of a normally distributed value (Figure 1.4) while the Heat delete operation time is a good example of a non-normal distribution with clustered response times (Figure 1.5). There are several methods for verifying the normal distribution hypothesis; these are described later in Section 1.4.2.

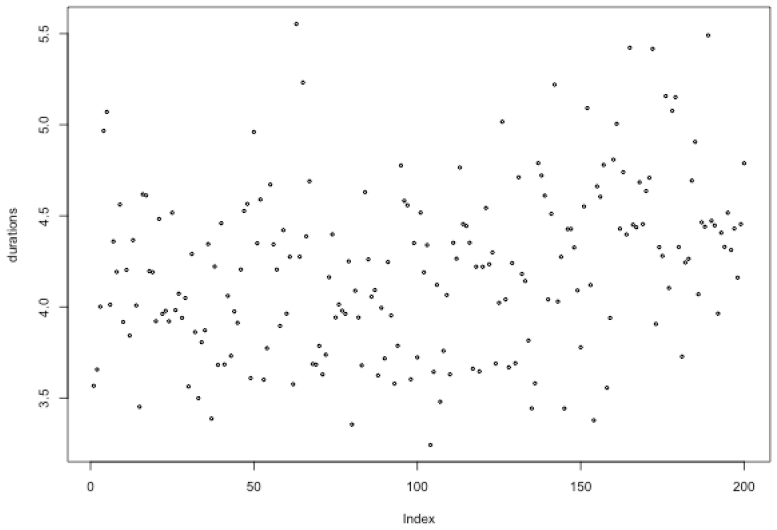



Figure 1.4: Heat create_stack operation response time deviation

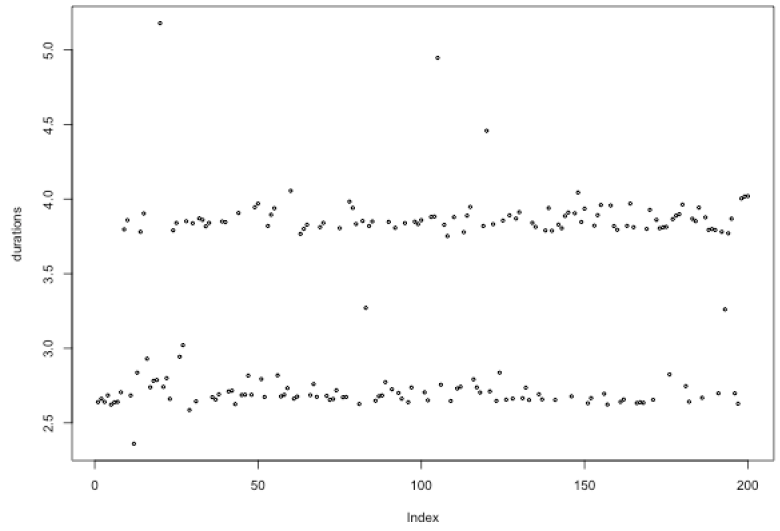



Figure 1.5: Heat delete_stack operation response time deviation

### 1.3.3 Non-normal Distributions

If the measured value distribution is far from normal, then either some dynamics of the system itself drives this departure, or that the measured value is under the influence of an underlying process which changes its value in a specific non-random way. A non-normal distribution requires deeper investigation of the measured system to understand what led to the unexpected distribution. For example, if the measured value is the request time for a value that nominally requires a database query but might be found in a system

cache, then the distribution will have two well distinguished peaks: one corresponding to the cache lookup time and another to the database query time in the event of a cache miss.

There are several approaches on how to deal with a non-normal distribution. In our experiments we have observed several types of such distributions:

- multiple, clearly distinguished peaks, Figure 1.6(a)
- wide bell shape with flat top, Figure 1.6(b)
- asymmetrical bell shape with long tails, Figure 1.6(c)

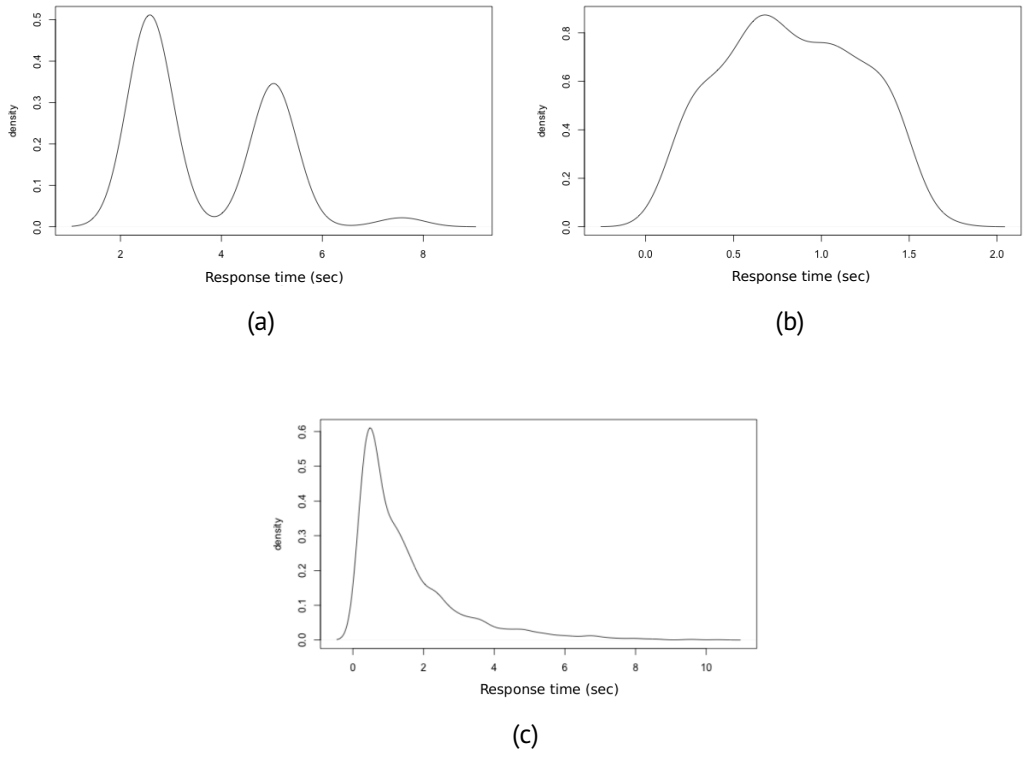



Figure 1.6: Examples of non-normal distributions (based on actual OpenStack test results)

Each type of distribution requires special handling and careful analysis. Several different root causes can lead to multiple peak distributions. It is possible to have multiple peaks when the system has several distinguished states where each is characterized by a different value. In this case, each peak is bell curved and can be described in terms of normal distribution parameters: mean value and standard deviation (Figure 1.6(a)). Such distributions are typical for clustered data and can be analyzed with various clustering methods. Clustering methods are discussed in Section 1.4.3.

A bell curve with flat top distribution can be caused by the coexistence of two processes with two characteristics and two significantly different frequencies corresponding to a slow process of value change or oscillations within some limited range and a fast

process of random influence (Figure 1.6(b)).

An asymmetrical bell curve distribution can be in fact be a Poisson distribution, which means that in this particular situation the system produces a significant number of random events much larger than the mean (Figure 1.6(c)).

## 1.4 Statistical Analysis Procedures

### 1.4.1 Stationarity Verification Method

As was mentioned in Section 1.4.1, stationarity is a statistical property or characteristic where the joint probability distribution does not change over time. As a consequence, statistical parameters like mean, variance, and other moments also do not change over time. This is the strict form of stationarity. The weak form of stationarity requires only that the first moments (mean, autocorrelation) do not vary over time, and is the form usually used.

There are several methods available to verify stationarity. The first one is the Moving Average model. This approach simply uses a moving window for the time series and for each window a mean value is calculated. The mean value behavior of different windows can be analyzed. If the mean values are localized within the acceptance range, then the process is considered a stationary process. If the mean value is changing, then the process is non-stationary. For the numerical evaluation of the mean value it is possible to use the following approach:

- split the whole time series into windows with some window size
- for each window $i$ calculate the mean value $\mu_i$
- take the first window mean value $\mu_0$ as a reference
- calculate the differences between first window mean value and the current window mean value $\Delta\mu_i = \mu_i - \mu_0$
- build a graph of the differences $\Delta\mu_i$ vs. $i$
- calculate an autocorrelation function for $\Delta\mu_i$

For a stationary process, the differences $\Delta\mu_i$ between mean values will be near $0$ and have close to a normal distribution. If the distribution is close to normal, the autocorrelation function will rapidly decay from significant values at small time lags to insignificant values at large lags.

For a non-stationary process, the differences $\Delta\mu_i$ will have the same trend as the original time series but with a smaller spread as some deviations will be filtered out by the averaging process. Since the trend still exists, the autocorrelation function will decay slowly with significant values remaining at large time lags. Both situations are illustrated in Figure 1.7.

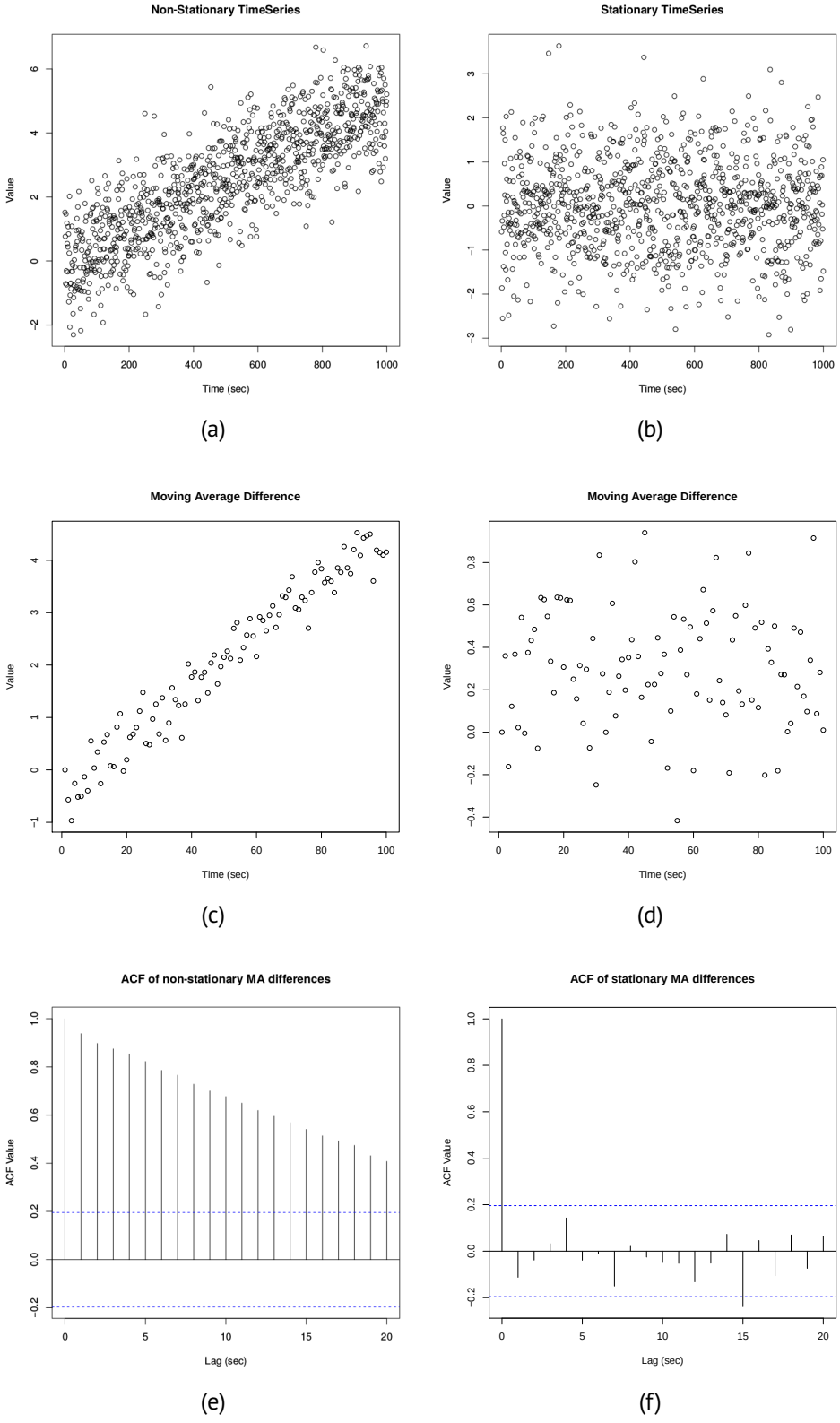



Figure 1.7: Moving Average method for non-stationary data (left) and stationary data (right)

A more sophisticated method is the ARMA model where the moving average approach is combined with an autoregression model. An Autoregression model is used in the Dickey-Fuller test for stationarity which by default checks for the existence of a trend. Sometimes it is preferable to test for stationarity when the null hypothesis is that the process is stationary. The Kwiatkowski, Phillips, Schmidt, and Shin (KPSS) test was designed to perform stationarity testing as well as to check other hypotheses. All of these tests are available in the **R** language and can be used for automated time series validation.

Once the trend is identified, the next step is to analyze the actual model of the underlying deterministic relationship of the parameters. For that task a regression analysis is a good start. In the case of a linear trend, a simple linear regression model can be used. For a non-linear trend, the regression task is difficult as preliminary information about the model should be known to do the analysis. In the OpenStack system most of the trends are linear and there is little need for complex non-linear regression models. Again, the **R** language provides regression model testing out of the box and can be used for time series analysis automation.

### 1.4.2 Normal Distribution Verification

### 1.4.3 Clustering Algorithm

As described in Section 1.3.3, some data sets may not be normally distribution, and the deviation from normal usually says something about either the system dynamics or about how the measured value is dependent on some underlying process. The most common type of non-normal distribution that we saw during performance and scalability testing is the multi-peak distribution. The Cinder delete volume operation is a good example of such a distribution (Figure 1.8).

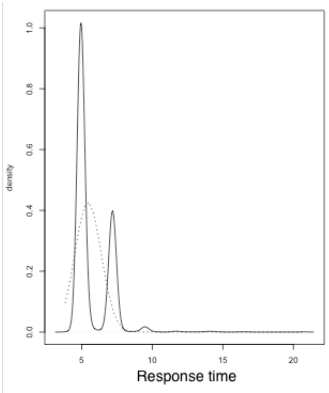



Figure 1.8: Distribution of duration of the Cinder delete volume operation [24]

In that case it will be useful to separate a full data set into smaller ones depending on the response duration distribution to analyze why the system behaves the way it does. This is commonly known as a cluster analysis task or clustering, which assumes that it is

possible to group objects inside a data set in a way that the objects in the same group cluster[2] are more similar (in terms of some specific criteria) to each other than to the objects in a different group.

This can be accomplished by various data mining methods, although it is difficult to find the ideal one that will be able to handle different density patterns and identify both multi-peak and the other patterns described above. Currently, we use two methodologies to separate a given data set into clusters:

- **Partition Around Medoids (PAM)** clustering algorithm (more precisely its PAMk variation)
- clustering approximation via density function extremes analysis

PAM is close to the k-means algorithm, as both of them use a data set partition in the root with error minimization, but the PAM algorithm uses medoids instead of centroid-based k-means. Medoids are close to centroids; the difference is that medoids are always a part of the original group of objects and can be used even if the mean or centroid cannot be defined.

The PAM algorithm partitions the original data set into $k$ clusters; it uses both the data set and the number $k$ as inputs. This algorithm works with a matrix of dissimilarity, whose goal is to minimize the overall dissimilarity between the representants of each cluster and its members. Pure PAM requires the specification of $k$ (number of clusters) as a prerequisite for the algorithm usage; however that is not usually possible in complex systems with various operations tested on multiple system topologies. This is the reason to choose the PAMk variation of the algorithm. It performs partitioning around medoids clustering with the number of clusters estimated by the optimum average Silhouette width or the Calinski-Harabasz index. More information about different approaches and heuristics for clustering is available in [19] and [17].

**Remark**

In our tests we used only the silhouette width to determine the number of clusters.

Both of the methods operate with a measure of how tightly all the data in the cluster is grouped. The Duda-Hart test[6] is applied to decide whether there should be more than one cluster, as in some specific cases pure PAM cannot understand that it is processing a data set that should not be clustered.

The PAMk algorithm is built-in into the **R** *fpc* package and can be easily used on the raw data.

The second clustering technique used is clustering approximation via density function extremes analysis.

Let's imagine we have an abstract data set, and in two-dimensional space it looks like Figure 1.9.

[2] Used here in a statistical sense: a group of objects selected by some criteria

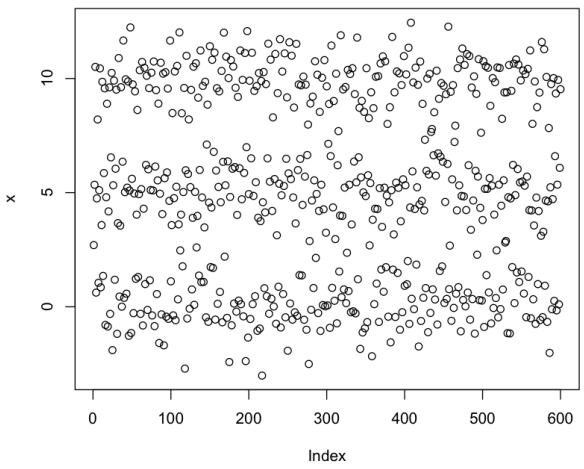



Figure 1.9: Clustered example data [24]

This data set has three clusters that clearly can be seen with the naked eye. If the size of data set is too large, it may be too expensive to use the PAM algorithm to analyze it due to its complexity. In this case we can build a density function of the distribution and analyze its extremes.

The density function of the given data is shown in Figure 1.10.

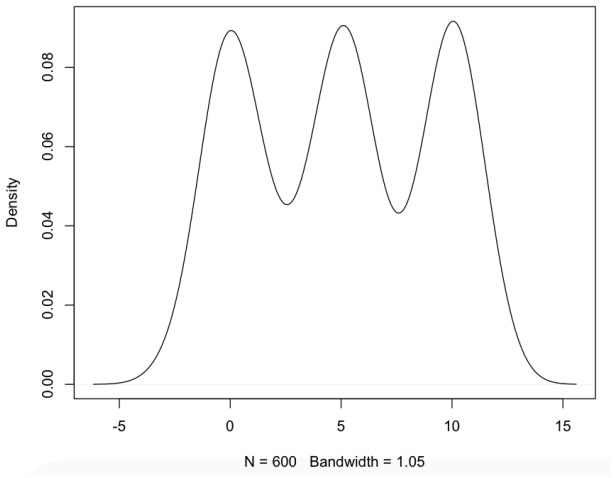



Figure 1.10: Density function of the example data [24]

The given data set can be split into three clusters – in fact this number is the same as the number of local maxima as we can clearly see in the chart above. In this particular case, the data set consists of three mixed normal distributions, which is why it is possible to split the original data set into three clusters as intervals between local minima like in Figure 1.11.

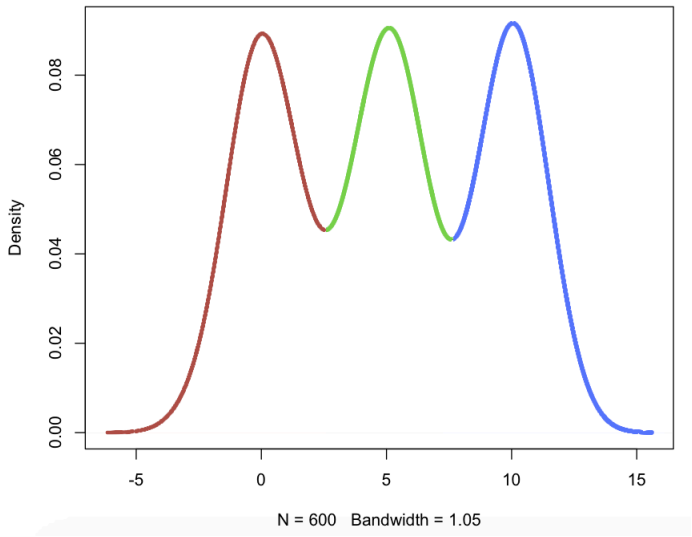



Figure 1.11: Density function of the example data, colored by cluster [24]

However, it is unlikely that such a clear pattern would be seen in real data. Even if the data were normally distributed, its clarity may be affected by different kinds of outliers. If the distribution does not look normal, the influence of outliers and long tails on the overall data appearance is even more significant.

This is the reason to sanitize the input data set before running any clustering algorithm or its approximation.

If we go back to Figure 1.3, it can be seen that the density of the standard normal distribution also has tails (probably less visible than it will be in real life distributions), and as it was mentioned earlier 95% of all values will be within $2\sigma$ of the mean value. We can sacrifice the remaining 5% to gain a clearer bell pattern and determine if the distribution is normal or not using only 95% of all values.

With real data it will be safe to cut the lower third of the density chart and work with only the upper two thirds. This will leave about 85% of the original data and will eliminate the influence of random outliers and possibly long tails.

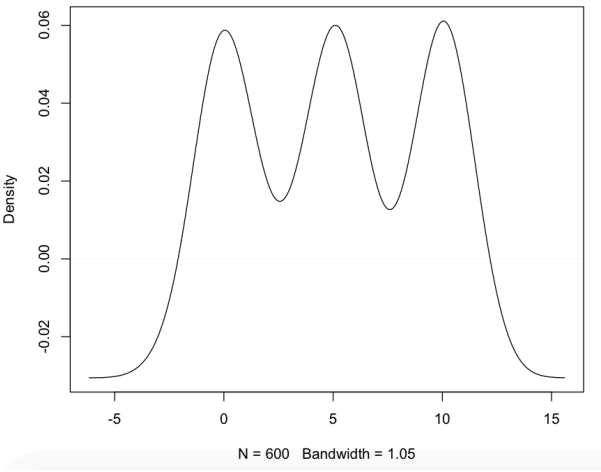



Figure 1.12: Density function of the example data (lower third cut) [24]

After this we can either use the PAM algorithm or approximate clusters as intervals constructed from the positive x axis values on the modified density chart. Its endpoints may be either intersections with the x axis or local minima.

## 1.5 Measuring Transition Processes

Any complex system can be either in a steady or transient state at a given moment in time. A steady state means that the system being analyzed has numerous properties that are unchanging in time. This means that for those properties $p$ of the system, the partial derivative with respect to time is zero:

$$\partial p/\partial t = 0 \tag{1.1}$$

In many systems, steady state is not achieved until some time after the system is started or initiated. This initial time interval is often identified as a transient state, start-up, or warm-up period. A newly created OpenStack cloud is not an exception here. This is why almost all performance measurements that might be run on an OpenStack cloud require measuring and then excluding the transition period from the main data analysis. Although transition periods are still interesting topics for separate research as they represent how the system performs when it is initially brought up and during failure recoveries and other disruptions to normal operations.

A good example of the transition process is presented in Figure 1.13. Glance (OpenStack Image Service) clearly demonstrates that the OpenStack cloud was in a transient state from the beginning of the test scenario. After about 150 iterations, the system became nearly steady (still, the process can not be called stationary since as more data was written to the database, the longer it took to extract this data, as evidenced by a slow growing pattern).

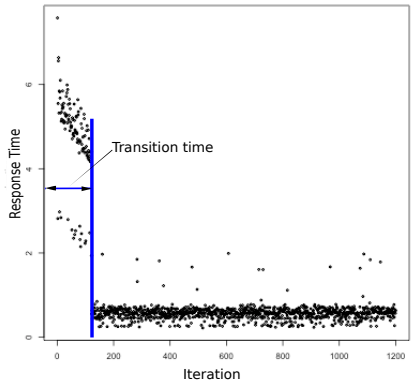



Figure 1.13: Transition process for Glance list images operation

Any transition process measurement assumes that we have information about the law the system works under. In the simplest case, if all data in a given data set are found to belong to a single cluster and verified to be stationary, the data set very likely has either a normal or a Poisson distribution. The data set therefore has certain characteristics. However, a transition process will not have all these characteristics, so such processes can be detected. Obviously this will not work for systems with multiple data clusters where each corresponds to a different stable system state. In this and other cases, the transition period analysis becomes much more complicated.

### 1.5.1 Why Analyze Transition Processes

In performance testing any transition process is usually undesired and most of the test methods require reducing the influence of the transition process on the measurement process. Although undesirable in most performance testing, the transition process is the primary object of interest for other tests. The transition process comprises the dynamics of the system in response to changed conditions or load. The system response to certain condition changes is often very important in predicting the behavior of the system in a production environment. The typical areas of interest are:

**HA** - system response in the event of failure of one of the components

**DR** - system response in the event of failure of a whole region in multi-region deployments

**DoS** - system response in the event of unusual load

**Burst Load** - system response in the event of high spikes of load

With an HA event, the characteristics of the transition period can be used for capacity planning and for defining SLA as well as verifying that system behavior meets certain SLA. The fact that HA dynamics depends on the amount of data, load, and utilization is one of the reasons why performance and scale testing is still relevant for transitional types of behavior.

Performance testing of an HA feature should be able to answer the following questions:

- What is the switchover time between HA components?
- How does the switchover time depend on the load?
- Is there a downtime during the switchover?
- How do the performance characteristics change during the switchover process and after it?
- Are there anomalies like performance drops or queue overflows and how do they affect performance?

For a DR scenario testing, it is important to validate the data loss expectations. Typical measurements are built around DR specific KPIs: Recovery Time Objective and Recovery Point Objective. DR solutions usually involve some kind of data replication across sites. This data replication can be in form of simple backup operations or more advanced DB log duplication with possibly synchronous or asynchronous DB replication. It can be provided by DB layers or by low level storage devices. As DR is not a typical system state, performance and scale characteristics of the DR process might drastically differ from the usual state of the cloud.

The last two items, DOS and Burst Load scenarios, are subjects of special interest and are designed to answer the question of how the system behaves when the network or other subsystem is under an unusually high load. This is a typical stress test which exposes the behavior of the system under extremely high loads. It is expected that the system will not crash and behave appropriately by providing correct responses like an HTTP 503 (Service Unavailable) Response code.


# 2. OpenStack Cloud

This chapter will focus on OpenStack cloud structure description and OpenStack services overview.

## 2.1 OpenStack Structure

OpenStack is designed as a distributed SOA[1] architecture which consists of multiple loosely coupled services. The level of coupling varies from service to service but in general the OpenStack approach is to avoid functionality duplication between services and keep dependencies between projects on a manageable level. OpenStack services communicate with each other via well defined API exposed by each service while most of the communication within service components is done via RPC[2] API through a message bus subsystem. Internally, OpenStack services are composed of several processes. All services have at least one API process, which listens for API requests, preprocesses them and passes them on to other parts of the service. With the exception of the Identity service, the actual work is done by distinct processes.

For communication between the processes of one service, an AMQP[3] message broker is used. The service's state is stored in a database. When deploying and configuring your OpenStack cloud, you can choose among several message broker and database solutions, such as RabbitMQ, Qpid, MySQL, MariaDB, and SQLite.

The logical schema of major OpenStack services is presented in Fig.2.1

[1] Service Oriented Architecture (SOA)
[2] Remote Procedure Call (RPC)
[3] Advanced Message Queuing Protocol (AMQP)

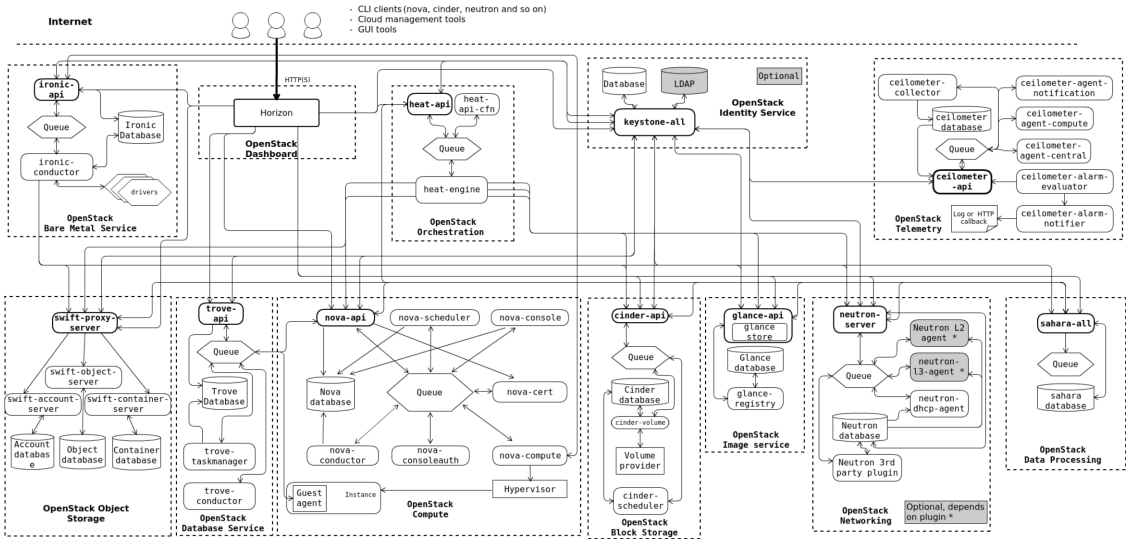



Figure 2.1: OpenStack logical architecture[7]

### 2.1.1 OpenStack Services Structure

#### OpenStack Compute

OpenStack Compute is responsible for hosting and managing cloud computing systems. OpenStack Compute is a major part of an IaaS[4] system. The main modules are implemented in Python.

OpenStack Compute interacts with OpenStack Identity for authentication, OpenStack Image service for disk and server images, and OpenStack dashboard for the user and administrative interface. Image access is limited by projects, and by users; quotas are limited per project (the number of instances, for example). OpenStack Compute can scale horizontally on standard hardware, and download images to launch instances.

OpenStack Compute components:

**nova-api** Accepts and responds to end user compute API calls. The service supports the OpenStack Compute API, the Amazon EC2 API, and a special Admin API for privileged users to perform administrative actions. It enforces some policies and initiates most orchestration activities, such as running an instance.

**nova-api-metadata** Accepts metadata requests from instances. The nova-api-metadata service is generally used when you run in multi-host mode with nova-network installations.

**nova-compute** A worker daemon that creates and terminates virtual machine instances through hypervisor APIs

**nova-scheduler** Takes a virtual machine instance request from the queue and determines on which compute server host it runs.

**nova-conductor** Mediates interactions between the nova-compute service and the database. It eliminates direct accesses to the cloud database made by the nova-compute

[4] Infrastructure-as-a-Service (IaaS)

service. The nova-conductor module scales horizontally.

**nova-cert** A server daemon that serves the Nova Cert service for X509 certificates. Used to generate certificates for euca-bundle-image. Only needed for the EC2 API.

**nova-network** Similar to the nova-compute service, accepts networking tasks from the queue and manipulates the network. Performs tasks such as setting up bridging interfaces or changing IPtables rules.

**nova-consoleauth** Authorizes tokens for users that console proxies provide. See nova-novncproxy and nova-xvpvncproxy. This service must be running for console proxies to work. You can run proxies of either type against a single nova-consoleauth service in a cluster configuration.

**nova-novncproxy** Provides a proxy for accessing running instances through a VNC connection. Supports browser-based novnc clients.

**nova-spicehtml5proxy** Provides a proxy for accessing running instances through a SPICE connection. Supports browser-based HTML5 client.

**nova-xvpvncproxy** Provides a proxy for accessing running instances through a VNC connection. Supports an OpenStack-specific Java client.

**nova-cert** x509 certificates.

OpenStack Compute service uses the following underlying infrastructure components and services:

**message-queue** A central hub for passing messages between daemons.

**SQL database** Stores most build-time and run-time states for a cloud infrastructure, including: Available instance types, Instances in use, Available networks, Projects

#### OpenStack Block Storage

The OpenStack Block Storage service (Cinder) adds persistent storage to a virtual machine. Block Storage provides an infrastructure for managing volumes, and interacts with OpenStack Compute to provide volumes for instances. The service also enables management of volume snapshots, and volume types.

The Block Storage service consists of the following components:

**cinder-api** Accepts API requests, and routes them to the cinder-volume for action.

**cinder-volume** Interacts directly with the Block Storage service, and processes such as the cinder-scheduler. It also interacts with these processes through a message queue. The cinder-volume service responds to read and write requests sent to the Block Storage service to maintain state. It can interact with a variety of storage providers through a driver architecture.

**cinder-scheduler** Selects the optimal storage provider node on which to create the volume. A similar component to the nova-scheduler.

**cinder-backup** The cinder-backup service provides backing up volumes of any type to a backup storage provider. Like the cinder-volume service, it can interact with a variety of storage providers through a driver architecture.

Block Storage service uses MQ service for communication between Cinder components.

#### OpenStack Networking

OpenStack Networking allows you to create and attach interface devices managed by other OpenStack services to networks. Plug-ins can be implemented to accommodate different networking equipment and software, providing flexibility to OpenStack architecture and deployment.

It includes the following components:

**neutron-server** Accepts and routes API requests to the appropriate OpenStack Networking plug-in for action.

**neutron-plugin** Plugs and unplugs ports, creates networks or subnets, and provides IP addressing. These plug-ins and agents differ depending on the vendor and technologies used in the particular cloud. OpenStack Networking ships with plug-ins and agents for Cisco virtual and physical switches, NEC OpenFlow products, Open vSwitch, Linux bridging, and the VMware NSX product.

**neutron-agent** The common agents are L3, DHCP, and a plug-in agent.

Networking service uses MQ service for communication between Neutron components.

#### OpenStack Identity

The OpenStack Identity service performs several functions: tracking users and their permissions, providing a catalog of available services with their API endpoints.

Each service in your OpenStack installation should be registered in Keystone service catalog as this is the place where all OpenStack clients are looking for API endpoints. Identity service can then track which OpenStack services are installed, and where they are located on the network.

OpenStack Identity service consists of a single service called keystone-api In typical deployments, OpenStack Identity service uses Memcache layer to improve user token management performance as well as allow HA[5] architecture.

#### OpenStack Image

The OpenStack Image service is one of the central infrastructure components which provides a storage and management for VM images and snapshots. It accepts API requests for disk or server images, and image metadata from end users or OpenStack Compute components. It also supports the storage of disk or server images on various repository types, including OpenStack Object Storage.

A number of periodic processes run on the OpenStack Image service to support caching. Replication services ensure consistency and availability through the cluster. Other periodic processes include auditors, updaters, and reapers.

The OpenStack Image service includes the following components:

**glance-api** Accepts Image API calls for image discovery, retrieval, and storage.

**glance-registry** Stores, processes, and retrieves metadata about images. Metadata includes items such as size and type.

OpenStack Image service uses the following underlying infrastructure components and services:

[5] High Availability (HA)

**Storage repository** Various repository types are supported including normal file systems, Object Storage, RADOS block devices, HTTP, and Amazon S3. Note that some repositories will only support read-only usage.

**SQL database** Stores image metadata and you can choose your database depending on your preference.

#### OpenStack Orchestration

The Orchestration module provides a template-based orchestration for describing a cloud application, by running OpenStack API calls to generate running cloud applications. The software integrates other core components of OpenStack into a one-file template system. The templates allow you to create most OpenStack resource types, such as instances, floating IPs, volumes, security groups and users. It also provides advanced functionality, such as instance high availability, instance auto-scaling, and nested stacks. This enables OpenStack core projects to receive a larger user base.

The service enables deployers to integrate with the Orchestration module directly or through custom plug-ins.

The Orchestration module consists of the following components:

**heat-api** An OpenStack-native REST API that processes API requests by sending them to the heat-engine over Remote Procedure Call (RPC).

**heat-api-cfn** An AWS Query API that is compatible with AWS CloudFormation. It processes API requests by sending them to the heat-engine over RPC

**heat-engine** Orchestrates the launching of templates and provides events back to the API consumer.

#### OpenStack Telemetry

The Telemetry module provides an opportunity to reliably collect measurements of the utilization of the physical and virtual resources comprising deployed clouds, persist these data for subsequent retrieval and analysis, and trigger actions when defined criteria are met. Its mission is to provide simple API for the metering and monitoring data, notifications collection, alarms triggering based on the collected data. Every data point is stored with projects and users information.

Telemetry Module is widely used as a auto-scaling backend for the Orchestration module, allowing to use changes to all collected data as an alarm trigger, that initiates HTTP(s) callback to the Orchestration module.

Telemetry module is also used for billing purposes as it contains history of all billable events inside the cloud like objects creation, updating, deletion, and status change.

The Telemetry module consists of the following components:

**ceilometer-api** An OpenStack-native REST API service that processes data read and write operations. In case of read or write operations, ceilometer-api communicates directly with storage abstraction layer connected to the chosen DataBase backend.

**ceilometer-collector** A service responsible for collecting and writing metering data coming from the ceilometer-anotification service to the storage backend used.

**ceilometer-apolling** A service responsible for measurements polling from different OpenStack services. Can either use OpenStack services API for data collection or compute hypervisors API to extract data about the instances running on the compute nodes. Can also collect power/thermal data from the IPMI sources. In the most common way all the work can be done by only one agent installed, but usually this agent is run with different parameters on different cloud nodes to achieve maximum performance. So ceilometer-apolling can be run as ceilometer-acompute, ceilometer-acentral, or ceilometer-aipmi depending on the polling parameters passed.

**ceilometer-acompute** A ceilometer-apolling service optimized to work on the compute nodes to collect instances-specific information from the compute node the agent is running on.

**ceilometer-acentral** A ceilometer-apolling service optimized to work on the cloud controllers nodes to poll OpenStack projects APIs.

**ceilometer-aipmi** A ceilometer-apolling service optimized to work on the cloud nodes and collect power/thermal data.

**ceilometer-anotification** A Telemetry service responsible for notifications coming from other OpenStack services processing and moving them to the format the ceilometer-collector service can understand

**ceilometer-alarm-notifier** A service that consumes triggers from ceilometer-alarm-evaluator and sends the alarms in the HTTP(s) callback format to the external system interested in it.

**ceilometer-alarm-evaluator** A service that evaluates that the metering data coming triggers the alarm in accordance with the predefined rules.

#### OpenStack Data Processing

OpenStack Data Processing module aims to provide cloud users with a simple way to provision and use data processing tools clusters (Hadoop, Twitter Storm, Spark) in OpenStack by specifying just several parameters (in case of Hadoop this is version, cluster topology, node hardware details and a few more).

The OpenStack Data Processing module supports various Hadoop distributions: Cloudera, Hortonworks, MapR, as well as Vanilla Apache Hadoop.

The OpenStack Data Processing module supports Elastic Data Processing (EDP) that allows to specify the data processing jobs to be run on the provisioned cluster.

The OpenStack Data Processing module consists of the following components:

**sahara-api** An OpenStack native REST API service that provides single endpoint for the external systems to communicate with. sahara-api translates REST API calls to the sahara-engine component in the form it can understand.

**sahara-engine** A service responsible for all the data processing related operations performing, such as data processing tools clusters deployment above OpenStack and EDP workloads.


# 3. OpenStack Control Plane Testing

## 3.1 API Performance Testing

OpenStack cloud interacts with end users via public cloud API. Almost all OpenStack services expose REST API. As OpenStack cloud provides services for its users, it is important to be able to define API Service Level Agreement (SLA) for each OpenStack component. SLA definition is discussed in ITIL [16] and ISO20000 [14]. There are common metrics which are used to define service level objectives:

**ASA** Average time (usually in seconds) it takes for a request to be answered. Also known as response time.

**TSF** Percentage of requests answered within a definite timeframe, e.g., 80% in 20 seconds

**MTBF** Mean Time Between Failures

**MTBSI** Mean Time Between Service Incidents

**MTRS** Mean Time to Restore Service

**TAT** Time taken to complete a certain task

**ER** Error rate. Number of requests for which service returned an error response

**RPS** Requests per second. A rate at what service can successfully handle requests.

Control Plane API testing aims to reveal actual reference numbers for these metrics to provide meaningful information about performance and scalability limitations of OpenStack services.

As it was discussed in Chapter 1, there are different types of measurements which can be performed. In this section we will discuss the actual measurable values which can be collected in direct and indirect measurements.

The following metrics can be directly measured:

**TAT** turnaround time is measured in form of direct measurement of service response time for each specific operation request. In our tests, the response time is measured as the time of receiving of the whole response body and the time to deserialize it so that a corresponding OpenStack client returns actual Python objects as a result. This is the cumulative time of request and response transmission over the transport level as well as the time of serialization and deserialization processes on both client and service sides plus the time service spends on processing the request. In our measurement this value is called $T_{resp}$

**ER** Error rate can be directly measured by processing the actual service response and verifying the response code. OpenStack API conforms to HTTP protocol specification[28] and uses standard HTTP response codes to specify the response status.

**Concurrent requests** the number of concurrently issued requests is controlled by testing framework and can be easily collected or set up to a specific value.

**CPU usage** CPU usage is an important metric which allows to evaluate the actual compute resources usage by any particular service. The test framework should provide an ability to measure CPU usage on the hosts where the service is running.

**Memory usage** Memory usage should be collected for each service process. This value is one of the key values for the capacity planning task.

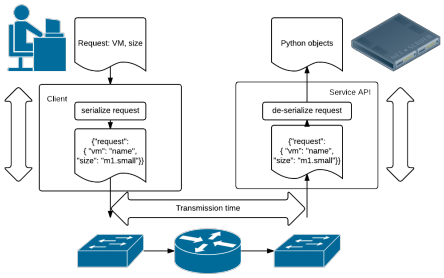



Figure 3.1: Request time components

These directly measured values can be used to perform indirect measurements for other important service performance metrics. Before we proceed with indirect measured values let's discuss how these indirect measurements could be performed and what kind of mathematical apparatus can help to perform this task.

### 3.1.1 API Operation Response Time Measurement

OpenStack API for each service is different but in general can be categorized into several categories:

- CRUD operations on service specific objects (instances, security groups, volumes, networks)
- LifeCycle operations like instance suspend, resume, assign floating address or creating a snapshot of an instance
- Service function operations like Identity token generation and validation

For each OpenStack service and for each specific API call specific test suites should be created. Each test suite is designed to perform a specific value measurement in order to provide information about scale and performance of the particular subsystem of the service.

In particular, for each API operation a response time should be measured. The actual measurement method could be specific to a service, but in general it should follow the procedures listed below:

**Important**

- Each measurement should be performed in the environment with a known state of the service and system in general. The proper "warm-up" period should be selected for each particular measurement by using preliminary analysis of the value graphs collected during a sufficient period of test run time.
- The actual measurement should not be affected by other requests to the service which is currently tested as well as to other services which might affect the measurement process. This means that concurrency should be equal to 1 and there are no other tests running against the same system. The effect of concurrency should be a subject of a separate scalability test.
- In order to obtain statistically reliable results, a measurement should be performed several times to obtain enough statistical information for the measured value.
- Stationarity of the measured value should be checked. The source of non-stationarity should be explored and if the non-stationarity is not introduced by measurement process, the special approach for non-stationary data should be used. The proper technique for the non-stationary data was discussed in the Chapter 1
- The hypothesis of standard distribution for the measured value should be checked. T-test of F-criteria can be used for that as well as more sophisticated validation can be used. The list of available normality test criteria is available in section 1.4.2
- In case of normally distributed values it is possible to use $1^{st}$ momentum as an actual value of the measured performance value and standard deviation can be used as a confident interval as it was discussed in section 1.3
- For non-standard distributions of the measured values, additional experiments and measurements should be performed.

The following format of the response time measurements should be used:

| API Operation | Average Response Time $T_{resp}$ (sec) | 95%ile (sec) |
|---|---|---|
| | | |

Table 3.1: OpenStack Compute API operation performance table format

### 3.1.2 Derived Values Measurement

For indirect measurements the source of the data values should be specified and computational formula should be also documented. The confidence interval for the indirect

(derived) measured value should be assessed by calculation of the systematic error induced by the measurement error of the directly measured value. The Law of Propagation of Errors can be used for the systematic error assessment[21]. If the formula for derived or indirect measurement is available, the variance method can be used [21].

Instead of systematic error a relative error can be used $CV_p = \sigma/p$ where $\sigma^2 = Var(p)$ is a variation of the measured value $P$. For the multiplicative forms of relation between derived value and directly measured values, the relative error of derived value is a sum of relative errors of measured values.

$$v = \frac{x * y * z}{p * q * r} \tag{3.1}$$

$$(CV_v)^2 = (CV_x)^2 + (CV_y)^2 + (CV_z)^2 + ... + (CV_p)^2 + (CV_q)^2 + (CV_r)^2 \tag{3.2}$$

**Little's Law**

The directly measured value of the response time allows to calculate specific capacity characteristics if there is additional information about the amount of worker processes/threads which will perform the API request handling. Assuming that a single worker can handle only one request at the time (which is true most of the time for OpenStack service) one can use the following formula to calculate the estimation of number of requests per second (RPS) which compute service can handle.

$$X_{rps} = N_{workers} / < T_{resp} > \tag{3.3}$$

This formula was introduced in 1960 and later became known as Little's Law [23]. Here $X_{rps}$ is a service throughput, $N_{workers}$ is a number of service workers and $< T_{resp} >$ is an average response time. This RPS value can be used as an estimate for the service throughput.

**Throughput Measurement Example**

For the API function VM create the only directly measurable parameter is a response time. It is possible to derive a throughput parameter by measuring the response time and using Little's law for the known amount of concurrently processed requests (by the number of workers). The following formula will be used for the throughput $X_{rps}$ estimation:

$$X_{rps} = N / T_{resp} \tag{3.4}$$

Where $N$ is a number of concurrently processed requests estimated by a number of process workers and $T_{resp}$ is a response time estimated as an average value obtained as a result of the direct measurement of the response times.

The variation of the equation 3.4 gives us an estimation of the systematic error as the following:

$$\delta X = -\frac{N}{T^2}\delta T \tag{3.5}$$

$$\sigma_{\delta X} = |-\frac{N}{T^2}|\sigma_{\delta T} \tag{3.6}$$

The confidential interval is provided by standard deviation of the systematic error calculated from the random errors of the measured value $T_{resp}$. The equation 3.6 gives a relation between the standard deviation of the $X_{rps}$ and the standard deviation of the directly measured value $T_{reps}$. The following format of the derived values should be used:

| Operation | $< T_{resp} >$ (sec) | Std.Dev. $\sigma_T$(sec) | Throughput $X_{rps}$ | Std.Dev. $\sigma_X$ |
|---|---|---|---|---|
| | | | | |

Table 3.2: OpenStack Compute API operation performance and throughput table format

### 3.1.3 Concurrency Testing

Concurrency testing is used for the finding of the non-trivial behavior of the system under normal load. Concurrent load can trigger the known or unknown processes that can happen only when the same resource is being accessed concurrently. The usual suspects for non-trivial behavior are:

- code which issues SQL queries to the database
- code which uses some shared resource: memory, logging subsystem, message bus
- code which uses storage for keeping the accessed data

The issues and limitations revealed by concurrent tests usually affect the system performance. Concurrency issues can significantly reduce the system performance and sometimes can lead to data corruption. That is why it is important to perform concurrent tests and reveal the effects of concurrency.

From the statistical perspective, concurrency issues in form of bottlenecks and locks lead to a significant increase of response times for all or for the selected requests. In a complex system with multiple participants in the data processing process it is usual to see the lock effect for some period on time, depending on the relative performance of the individual components.

The following approach should be used for the concurrence testing:

**Important**

- The same tests which were used in the value measurement tests should be used.
- The concurrency parameter which controls the number of the concurrently issued requests should be varied in a reasonable range in order to measure the effects of the concurrency on different loads.
- The following graphs should be collected:
  1. Response time over time
  2. Average response time vs. number of concurrent requests
  3. Maximum response time vs. number of concurrent requests
  4. Relative number of response time to a single measured response time vs. the

number of concurrent requests
5. Number of error responses vs. number of concurrent requests
6. MTBF vs number of concurrent requests. Measured as time difference between subsequent Error responses.

### 3.1.4 Transition Process Measurement Method

Transition process is a subject of special interest of DevOps and architect teams. It is important to understand the dynamics of the system reacting to a specific event. In a complex system like an OpenStack cloud, multiple instances of the service processes are running concurrently and are distributed across a physical infrastructure. They are supported by using different infrastructure components like message buses, load balancers, DB clusters etc. From the performance and scale perspective of an OpenStack cloud, the subjects of the particular interest are:

- system reaction to the service process failure (HA)
- system reaction to the underlying infrastructure component failure (HA)
- system reaction to the whole site failure (DR)

The transition process testing requires special framework and setup as the process is tightly bound in time and proper synchronization between testing components is required.

In section 3.1 there were several metrics introduced which provide a meaningful picture of the transition process from the SLA perspective.

The failure test which simulates the failure of one of the service processes is designed to provide information about system performance and behavior in an HA scenario. The test is organized in the following way:

**Important**

- Standard performance measurements started as described in section 3.1.1
- After the warm-up period is passed and the measured values are steady, the initial measurement is done for the initial state definition. These numbers will be used as a reference in further test steps.
- a service failure is simulated by killing a service process or by shutting down the entire physical node.
- All the changes in the measured values are recorded for each period of time (1-5) seconds
- As soon as all measured values become steady, the new values are recorded as a definition of the performance at degraded state.
- If an error rate is not zero, the MTRS value is approximated by the time period when ER is non-zero.
- the transition period time value is approximated as a time difference between the time value, when all measurements are stabilized, and the time of the service failure

## 3.2 Density Testing

Density testing is used to define the capacity limits of the OpenStack cloud. The test method uses a high amount of created objects without introducing high concurrency so that the object numbers are growing slowly under normal load. In a test like this it is difficult to define the acceptance criteria for the limit as the cloud is in a healthy working state. Typically these limits are introduced by measuring the performance metrics with a specified SLA and the number objects at the moment of SLA criteria failure. The typical density test result graph is shown in Fig.3.2.

3.2

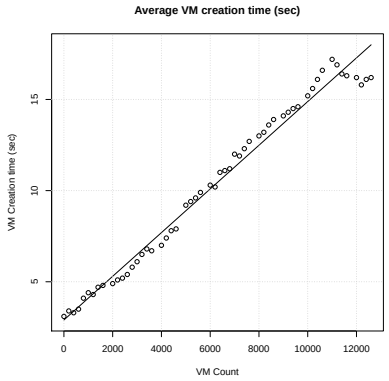



Figure 3.2: Nova API VM create response times over number of existing VMs

In this example we can flexibly define our capacity limits by defining our desired SLA. For the SLA for a VM create operation equal to 10 seconds the corresponding VM capacity limit will be around 6000 VMs existing in the cloud. Density testing methodology consists of the following steps:

**Important**

- Defining the SLA values as acceptance criteria
- Start adding a load to the system. Proper concurrency values should be used to generate a steady and non-breaking load
- Measure metrics values and compare them to the defined SLA
- As soon as the measured metrics do not conform to the SLA criteria the load should be removed from the system. The actual number of the created objects is used as limit approximation
- The system should be kept in the same state for a period of time to validate that this load limit still allows the system to function properly

### 3.2.1 Detailed Performance Testing Plan

**Keystone**

The current version of Keystone API is Version 3. This version introduces new concepts and API for domains and trusts. Domains represent collections of users, groups, and

projects. Each of them is owned by exactly one domain. Users, however, can be associated with multiple projects by granting roles to the user on a project, including projects owned by other domains.

Each domain defines a namespace where certain API-visible name attributes exist, which affects whether those names must be globally unique or unique within that domain. In the Identity API, the uniqueness of the following attributes is as follows:

- Domain name. Globally unique across all domains.
- Role name. Globally unique across all domains.
- User name. Unique within the owning domain.
- Project name. Unique within the owning domain.
- Group name. Unique within the owning domain.

Other major parts of Keystone API remain the same as in Version 2. The complete list of API components and their URIs is in the table 3.3

| API Group | URI |
|---|---|
| Tokens | /v3/auth/tokens |
| Services | /v3/services |
| Endpoints | /v3/endpoints |
| Domains | /v3/domains |
| Projects | /v3/projects |
| Users | /v3/users |
| Groups | /v3/groups |
| Credentials | /v3/credentials |
| Roles | /v3/roles |
| Policies | /v3/policies |

Table 3.3: OpenStack Identity (Keystone) API components and their URI

#### Tokens

Tokens API operations are the most used API operations in OpenStack. Each OpenStack service uses Identity token API for authentication. For example, Nova VM create operation will do up to 15 calls to Identity token API for authentication and token validation. The following tests should be performed to identify the service behavior for Token API:

**Performance Test** Response time and TPS should be measured for a single operation. Token operation can have a different size of a response. This is controlled by the request options and if the call has no explicit authorization scope, the response does not contain the catalog, project, domain, or role fields. However, the response still uniquely identifies the user. A token scoped to a project also has both a service catalog and the user's roles applicable to the project. A token scoped to a domain also has both a service catalog and the user's roles applicable to the project.

**Concurrency Test** Find the effective number of parallel requests which the service can

| API Operation | Description |
|---|---|
| POST /v3/auth/tokens | Authenticates and generates a token |
| GET /v3/auth/tokens | Validates a specific token |
| HEAD /v3/auth/tokens | Validates a specific token |
| DELETE /v3/auth/tokens | Revokes a specified token |

Table 3.4: Keystone Tokens API operations and their descriptions

| API Operation | Performance | Concurrency | Scalability |
|---|---|---|---|
| POST | Value $T_{resp}$, Value $X_{TPS}$ | Graph $T_{resp}$ vs. $N_{concurrent}$ | Graph $T_{resp}$ vs. $N_{tokens}$, Value $max(N_{tokens})$ |
| GET | Value $T_{resp}$, Value $X_{TPS}$ | Graph $T_{resp}$ vs. $N_{concurrent}$ | Graph $T_{resp}$ vs. $N_{tokens}$, Value $max(N_{tokens})$ |
| HEAD | Value $T_{resp}$, Value $X_{TPS}$ | Graph $T_{resp}$ vs. $N_{concurrent}$ | Graph $T_{resp}$ vs. $N_{tokens}$, Value $max(N_{tokens})$ |
| DELETE | Value $T_{resp}$, Value $X_{TPS}$ | Graph $T_{resp}$ vs. $N_{concurrent}$ | Graph $T_{resp}$ vs. $N_{tokens}$, Value $max(N_{tokens})$ |

Table 3.5: Identity Token operations test types and expected measurement results

handle. Find the dependency between the number of parallel requests and service response time. Compare the results with the theoretical value obtained by applying Little's Law using a known number of workers. Assess the number of the concurrently processed requests by using measured values of service performance $T_{resp}$ and $X_{TPS}$. Because of using Greenlet/Eventlet a single worker can process multiple requests "almost" simultaneously. Thus, the number of Keystone workers is not exactly equal to the number of simultaneously processed requests.

**Scalability Test** Find the limit of objects in the service when the service is still responsive. Find the relation between the number of objects in the service (DB) and a response time for the API request.

The list of actual measured values for each token operation is presented in Table 3.5 while the meaning of each operation is explained in Table 3.4.

For the scalability test it is important to identify any trends which reflect the change of the system behavior when changing the load on the cloud. Please, refer to section 1.4.1 for analysis details.

**Token API: Authenticate**

The POST request to the Token API URI /v3/auth/tokens performs the authentication of the user by validating the supplied requests's information about user credentials. The body of the request must include a payload of credentials including the authentication method and, optionally, the authorization scope. The scope includes either a project or domain. If both project and domain are included, an HTTP 400 Bad Request results, because a token cannot be simultaneously scoped as both a project and a domain. If the optional scope is not included and the authenticating user has a defined default project (the **default_project_id** attribute for the user), that default project is treated as the preferred authorization scope. If no default project is defined, the token is issued without an explicit scope of authorization. One of the following sets of credentials should be provided to authenticate:

- user ID and password
- user name and password scoped by domain ID or name
- user ID and password scoped by project ID or name with or without domain scope, or token.

User ID and password, user name and password scoped by domain ID or name, user ID and password scoped by project ID or name with or without domain scope, or token. If the scope is included, project ID uniquely identifies the project. However, project name uniquely identifies the project only when used in conjunction with a domain ID or a domain name. A typical request body looks like the following:

```
{
    "auth": {
        "identity": {
            "methods": [
                "password"
            ],
            "password": {
                "user": {
                    "id": "0ca8f6",
                    "password": "secretsecret"
                }
            }
        },
        "scope": {
            "project": {
                "domain": {
                    "id": "1789d1"
                },
                "name": "project-x"
            }
        }
    }
}
```

If the request is successful, the Identity service will return a new token which can be

used for all further requests on behalf of the user. Each REST request requires inclusion of a specific authorization token HTTP x-header, defined as **X-Auth-Token**. Clients obtain **X-Auth-Token** and the URL endpoints for other service APIs by supplying their valid credentials to the authentication service.

The response to an authentication request returns the token ID in the **X-Subject-Token** header instead of doing so in the token data. If the call has no explicit authorization scope, the response does not contain the catalog, project, domain, or roles fields. However, the response still uniquely identifies the user. A token scoped to a project also has both a service catalog and the user's roles applicable to the project. A token scoped to a domain also has both a service catalog and the user's roles applicable to the project. Optionally, the Identity API implementation might return an authentication attribute to indicate the supported authentication methods. For authentication processes that require multiple round trips, the Identity API implementation might return an HTTP 401 Unauthorized error with additional information for the next authentication step. A typical simple successful response is presented below:

```
{
    "token": {
        "expires_at": "2013-02-27T18:30:59.999999Z",
        "issued_at": "2013-02-27T16:30:59.999999Z",
        "methods": [
            "password"
        ],
        "user": {
            "domain": {
                "id": "1789d1",
                "links": {
                    "self": "http://identity:35357/v3/domains/1789d1"
                },
                "name": "example.com"
            },
            "id": "0ca8f6",
            "links": {
                "self": "http://identity:35357/v3/users/0ca8f6"
            },
            "name": "Joe"
        }
    }
}
```

As the actual amount of information processed for the each Authenticate request depends on the supplied scope in the request, all possible combinations should be tested. For each specific combination the whole scope of values defined in Table 3.5 should be measured and analyzed.

#### Token API: Validation

GET and HEAD requests to the token auth URI performs validation of the existing token supplied in the request header. A service token should be supplied in the **X-Auth-Token** header and the token to be validated in the **X-Subject-Token** header. The Identity API returns the same response for the GET request as when the subject token is issued by POST /auth/tokens as it was discussed earlier. The following GET request represents the typical Token validation request.

```
1 GET /v3/auth/tokens HTTP/1.0
2 X-Auth-Token: 1dd7e3
3 X-Subject-Token: c67580
```

#### Token API: Revoke

DELETE request to the Token API URI /v3/auth/tokens will immediately revoke the token supplied in the request header **X-Subject-Token**. An additional **X-Auth-Token** is not required. The response will not contain any body and all the necessary information is passed via standard HTTP response codes.

#### Failover Scenario

As the token API is the central API which is used by all other services, the effective failover of this service is critical for the entire cloud functioning. If token API is not responsive, no cloud operations are possible even if all other services are working correctly. It is worth mentioning that VMs and workloads are not affected by the Identity service functionality if they do not use it.

Production deployment of the Keystone service assumes that its services instances will be deployed on different physical nodes and synchronization between service instances will be done via DB layer and distributed cache layer. A typical deployment schema is shown in Figure 3.3

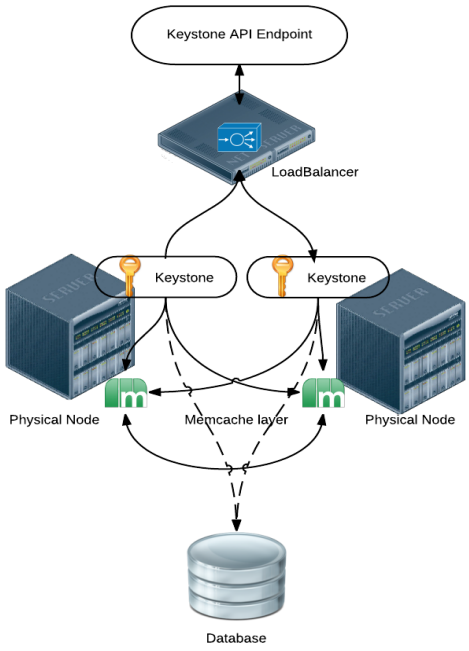



Figure 3.3: Keystone HA deployment example schema

Failover testing will reveal the behavior of the Identity service in the event of node failure. This testing will be mostly focused on the transition period evaluation to identify the SLA related metrics as well as to check the performance degradation. The failure test should be performed in a highly controlled environment where all components are under control of the testing framework. The ability to measure exact moments of the node failure is important. If there is no way to control the failure process, it is possible to use indirect methods like monitoring systems or logs stream for the measurements.

The following statistics will be collected:

| Token Operation | $< T_{resp} >$ HA | $< T_{resp} >$ after a failure | Transition time |
|---|---|---|---|
| Authenticate | | | |
| Validate | | | |
| Revoke | | | |

Table 3.6: Identity failover testing collected metrics

**Services**

Service catalog is used by almost all OpenStack clients and by some OpenStack services. OpenStack Orchestration – Heat service is a good example of the OpenStack service which uses a Service catalog. Service catalog operations are presented in Table 3.7

| API Operation | Description |
|---|---|
| POST /v3/services | Creates a new service |
| GET /v3/services | List existing services |
| GET /v3/services/{id} | Shows service details |
| PATCH /v3/services/{id} | Update a service |
| DELETE /v3/services/{id} | Delete a service entry |

Table 3.7: Keystone Service API operations and their description

Service API follows the standard REST API practices. For a specific entity operation a service ID should be provided as a part of a request URI. Create and List operations do not require a service ID specified while the update, read details and delete operations require a specific service ID. The following request and response bodies combination for a create operation is provided below.

```
{
    "service": {
        "type": "volume"
    }
}
```

```
{
    "service": {
        "enabled": true,
        "id": "686766",
        "links": {
            "self": "http://identity:5000/v3/services/686766"
        },
        "type": "volume"
    }
}
```

The same combination of measurements should be performed as it is done for the Token API.


# 4. OpenStack Data Plane Testing

## 4.1 Data Plane Testing Methodology

OpenStack Data Plane is an important part of overall information systems workability. In general, Data Plane covers any data operations performed by the workloads on top of OpenStack infrastructure. In particular, the most common parts are network operations which are responsible for data transfer over network protocols between VMs or between a VM and the external networks including Internet. Other parts of Data Plane layer are related to the storage operations which are used by the workloads to store data on a persistent storage. Together with the Control Plane and the Management Plane, these three are the basic components of a telecommunications architecture. The Data Plane enables data transfer to and from clients, handling multiple conversations through various protocols, and manages communications with remote peers.

### 4.1.1 Network Performance Testing

Measuring network performance has always been a difficult and loosely defined task, mainly because performance engineers are unclear with type of loads they need to test and what methods will be most appropriate for a specific network configuration.

A common (and very simple) method of network performance testing is by initiating a simple file transfer from one end (client) to another (server), however, this method is frequently debated amongst engineers and there is good reason for that: when performing the file transfers, we are not only measuring the transfer speed but also hard disk delays on both ends of the stream. It is very likely that the destination target is capable of accepting greater transmission rates than the source is able to

send, or the other way around. These bottlenecks, caused by hard disk drives, operating system queuing mechanism or other hardware components, introduce unwanted delays, ultimately providing incorrect results.[22]

Networking performance testing requires special tools to minimize the effects of other subsystems on the test results. Actual tools for the network performance measurement are listed in chapter 6. This section is focused on the networking performance metrics and methods rather than a tools discussion. Before the actual methodology is discussed, it is necessary to introduce the terminology to avoid confusion and misunderstanding caused by loosely defined terms. In this section the terminology from [9] is used with some additional terms taken from different sources.

**Interface** The term (NIC) Interface port refers to the physical network connector. The term "interface" or "link" refers to the logical instance of a network interface port, as detected and configured by the operating system.

**Virtual Interface** VIF is an abstract virtualized representation of a computer network interface that may or may not correspond directly to a network interface controller. OpenStack networking service (Neutron) supports multiple different VIF types [25]. Virtual interfaces are usually software devices implemented by the hypervisor. To provide a hardware level performance, a physical network device can be exposed directly to the virtual machine through various technologies like SR-IOV and PCI-e Hardware support for device passthrough. A great overview of various technologies is available in on-line IBM publication [18]. OpenStack supports SR-IOV passthrough as described in [32]. Recently a more advanced hardware technology emerged like Cisco(tm) UCS Virtual Interface Card 1280 with up to 256 dynamic virtual adapters support with an almost hardware performance level [3]. Performance measurements for this device can be found in [35] with VMware hypervisor used.

**Packet** The term packet typically refers to an IP-level routable message.

**Frame** A physical network-level message. For example, an Ethernet frame.

**Bandwith** Bandwidth: the maximum rate of data transfer for the network type, usually measured in bits per second. “10 GbE” is Ethernet with a bandwidth of 10 Gbits/sec.

**Throughput** The current data transfer rate between the network endpoints, measured in bits per second or bytes per second.

**Latency** Network latency can refer to the time it takes for a message to make a round trip between endpoints, or the time required to establish a connection (e.g., TCP handshake), excluding the data transfer time that follows.

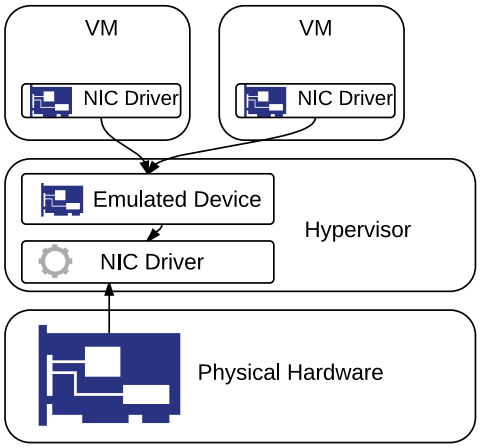



Figure 4.1: Virtual Interface connection layers

**Packet Size**

Packet size is usually limited by the network interface maximum transmission unit (MTU) size, which is configured to 1500 bytes for many Ethernet networks. Ethernet supports larger packets (frames) of up to approximately 9000 bytes, termed jumbo frames. These can improve network throughput performance, as well as latency of data transfers, by requiring fewer packets. Confluence of the two components has interfered with the adoption of jumbo frames: older network hardware and misconfigured firewalls. Older hardware that does not support jumbo frames can either fragment the packet using the IP protocol or respond with an ICMP "can't fragment" error, letting the sender know that they need to reduce the packet size. Now the misconfigured firewalls come into play: there have been ICMP-based attacks in the past (including the "ping of death"), to which some firewall administrators have responded by blocking all ICMP. This prevents the helpful "can't fragment" messages from reaching the sender and causes network packets to be silently dropped once their packet size increases beyond 1500. To avoid this issue, many systems stick to the 1500 MTU default. Performance of 1500 MTU frames has been improved by the network interface card features, including TCP offload and large segment offload. These send larger buffers to the network card, which can then split them into smaller frames using dedicated and optimized hardware. This has, to some degree, narrowed the gap between the 1500 and 9000 MTU network performance.

**Latency**

Latency is an important metric for the network performance and can be measured in different ways, including name resolution latency, ping latency, connection latency, first-byte latency, round-trip time, and connection life span. These are described as measured by a client connecting to a server.

**Name Resolution Latency** When connections are being established to remote hosts, a host name is usually resolved to an IP address, for example, by DNS resolution. The time this takes can be measured separately as name resolution latency. Worst case for this latency involves name resolution time-outs, which can take dozens of

seconds. Sometimes name resolution isn't necessary for the application to function and can be disabled to avoid this latency.

**Ping Latency** This is the time for an ICMP echo request to echo response, as measured by the ping command. This time is used to measure network latency between hosts, including hops in between, and is measured as the time needed for a packet to make a round trip. It is in common use because it is simple and often readily available: many operating systems will respond to ping by default.

**Connection Latency** Connection latency is the time to establish a network connection, before any data is transferred. For TCP connection latency, this is the TCP handshake time. Measured from the client, it is the time from sending the SYN to receiving the corresponding SYN-ACK. Connection latency might be better termed as connection establishment latency, to clearly differentiate it from connection life span. Connection latency is similar to ping latency, although it exercises more kernel code to establish a connection and includes time to retransmit any dropped packets. The TCP SYN packet, in particular, can be dropped by the server if its backlog is full, causing the client to retransmit the SYN. This occurs during the TCP handshake, so connection latency includes retransmission latency, adding one or more seconds.

**First-Byte Latency** Also known as time to first byte (TTFB), first-byte latency is the time from when the connection has been established to when the first byte of data is received. This includes the time for the remote host to accept a connection, schedule the thread that services it, and for that thread to execute and send the first byte. While ping and connection latency measures the latency incurred by the network, first-byte latency includes the think time of the target server. This may include latency if the server is overloaded and needs time to process the request (e.g., TCP backlog) and to schedule the server (CPU run-queue latency).

**Round-Trip Time** Round-trip time describes the time for a network packet to make a round trip between the endpoints.

**Connection Life Span** Connection life span is the time from when a network connection is established to when it is closed. Some protocols use a keep-alive strategy, extending the duration of connections so that future operations can use existing connections and avoid the overheads and latency of connection establishment.

### 4.1.2 Testing Methodology

#### Direct Use Method

The direct use method is for the quick bottlenecks and errors identification across all components. For each network interface, and in each direction — transmit (TX) and receive (RX) — check for:

**Utilization** the time the interface is busy sending or receiving frames

**Saturation** the degree of extra queueing, buffering, or blocking due to a fully utilized interface

**Errors** for receive: bad checksum, frame too short (less than the data link header) or too

long, collisions (unlikely with switched networks); for transmit: late collisions (bad wiring)

Errors may be checked first, since they are typically quick to check and are the easiest to interpret. Utilization is not commonly provided by operating system or monitoring tools directly. It can be calculated as the current throughput divided by the current negotiated speed, for each direction (RX, TX). Current throughput should be measured as bytes per second on the network, including all protocol headers. For the environments that implement network bandwidth limits (resource controls), as occurs in some cloud computing environments, network utilization may need to be measured in terms of the imposed limit, in addition to the physical limit. Saturation of the network interface is difficult to measure. Some network buffering is normal, as applications can send data much more quickly than an interface can transmit it. It may be possible to measure as the time application threads spend blocked on network sends, which should increase as saturation increases. Also check if there are other kernel statistics more closely related to the interface saturation, for example, Linux "overruns" or Solaris "nocanputs." An example of interface measurement report is provided below in table 4.1.

| Interface | Test Time $T$ (sec) | RX (bytes) | TX (bytes) | Utilization (%) |
|---|---|---|---|---|
| | | | | |

Table 4.1: Interface utilization measurements report

#### Workload Simulation

Characterizing the load applied is an important exercise during the capacity planning, benchmarking, and workloads simulation. It can also lead to some of the largest performance gains by identifying the unnecessary work that can be eliminated. The following basic attributes for network workload characterization can, together, provide an approximation of what the network is asked to perform:

**Network interface throughput** RX and TX, bytes per second
**Network interface IOPS** RX and TX, frames per second
**TCP connection rate** active and passive, connections per second

#### Latency Analysis

Latency analysis can be very useful to understand the network behavior. As it was discussed before, there are different latency types which can be measured. The simplest to measure is ping latency, but RTT latency can be also used. A single latency measure is not that useful, while latency analysis over some changes might be very productive. The following measurement types may add more value to the network behavior analysis.

**Latency vs Packet Size** the graph of latency change with the change of the transmitted packet size. This type of test can provide information about underlying network devices and their buffering and processing limitations. Some network devices use

ASICs to process network packets and these ASICs can have hardware limitations for the packet size.

**Latency vs Concurrent Connections** the graph of latency change with the change of the concurrent connections number. Most of the servers can handle a limited amount of concurrent connections. As soon as this limit is achieved, the system behavior can vary. Usually the latency value will be affected by operating system configurations (TCP backlog, SYN queue size, accepted connections queue) as well as server load handle strategy.

#### Performance Metrics Analysis

Performance monitoring can identify active issues and behavior patterns over time. It will capture variations in the number of active end users, the timed activity including distributed system monitoring, and application activities including backups over the network. Key metrics for network monitoring are

**Throughput** network interface bytes per second for both receive and transmit, ideally for the each interface

**Connections** TCP connections per second, as another indication of network load

**Errors** including dropped packet counters

**TCP retransmits** also useful to record for correlation with network issues

**TCP out-of-order packets** can also cause performance problems

For the environments that implement network bandwidth limits (resource controls), as occurs in some cloud computing environments, statistics related to the imposed limits may also be collected.

### 4.1.3 Shaker Tool

Shaker is an open source project which aims to perform data plane testing for the networking subsystem of the OpenStack. It uses Heat templates to deploy specific topologies of VM based traffic generators and then run tests and collect network specific statistics.

Shaker runs three types of network tests with many different options (including TCP and UDP). Below is a summary of the tests and their characteristics.

- type of the test:
  - VMs in the same network (L2)
  - VMs in a different network (L3 east/west)
  - VMs hitting external IP addresses (L3 north/south)
- communication: either floating IPs or SNAT/internal
- number of VMs: 1/10/20/50/100
- external hosts to use: static hard coded
- VM placement:
  - one VM per compute
  - two VMs per compute
  - two VMs per compute (different networks)

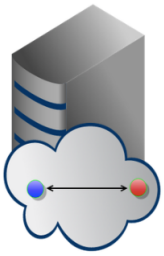


Figure 4.2: Shaker L2 test

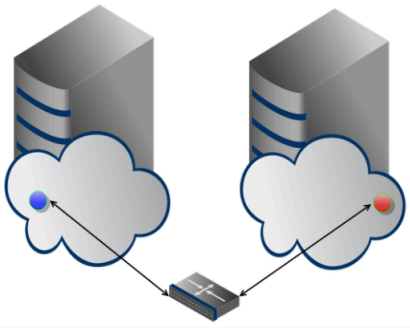


Figure 4.3: Shaker L3 east-west test

**Shaker L2 Segment Topology**
With VMs in the same network (L2 network test – Fig.4.2), Shaker deploys two VMs in the same network using Heat templates, and runs iperf between the two VMs, and measures the single stream network performance between the two VMs.

**Shaker L3 East-West Topology**
With VMs in different networks (L3 east/west - Fig.4.3), Shaker deploys two VMs in different networks using Heat templates, and runs iperf between the two VMs, and measures the single stream network performance between them. This will involve routing and will test the performance of the deployed SDN overlay.

**Shaker L3 North-South Topology**
The last case is about VMs hitting external IP addresses (L3 north/south - Fig.4.4). Shaker deploys one of the VMs with an external (floating) IP address, and runs iperf between the master node and the VM.

### 4.1.4 Performance Tests

For each networking topology discussed above Shaker collects specific information about network traffic. First of all, Shaker records actual time series for the measured values, so it is possible to perform a statistical analysis of the data as well as observational analysis. Shaker collects the following time series data:

- Data Upload Throughput value for individual worker

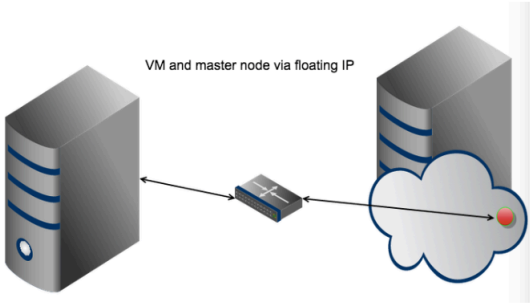



Figure 4.4: Shaker L3 north-south test

- Data Download Throughput value for individual worker
- Round Trip Time - Latency for individual worker
- Average Data Upload Throughput value over all workers
- Average Data Download Throughput value over all workers
- Average Round Trip Time - Latency over all workers

These values could be collected for both TCP and UDP based traffic to distinguish latency introduced by the protocol processing. Individual worker data allows to validate the networking connection behavior for the individual VM to check the effectiveness of networking traffic distribution in the network as well as to find possible short term discrepancies in the network connection.

Average values over all workers allow to evaluate the network performance of the cloud overall. Comparison of the different topologies results can highlight the actual behavior of the network on different levels.

Figures 4.5 and 4.6 show the typical time series for the individual VM which participates in the testing. Time series usually represent raw data and are used as a source for further statistical analysis.

As discussed in chapter 1, the initial warming up time is usually required. As shown in Figure 4.5, the initial period within several seconds should be excluded from the statistical analysis as not representative. During this period test tools set up its workers and initiated connections before sending actual data.

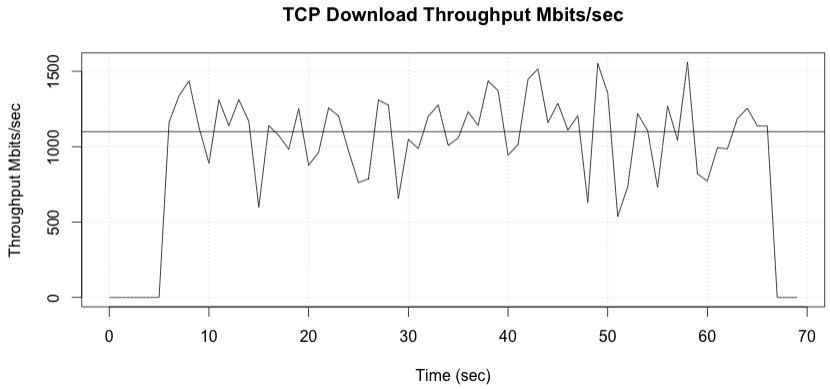



Figure 4.5: Throughput time series for the individual worker (VM) in L2 test

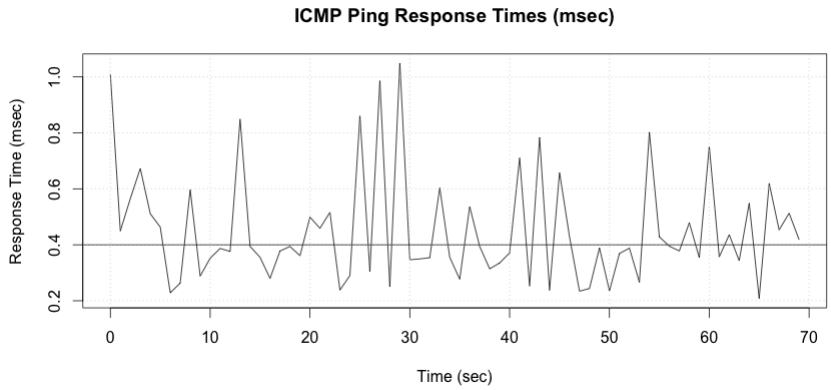



Figure 4.6: Ping latency time series for the individual worker (VM) in L2 test

Simple statistical analysis can provide important information like average throughput which can be used for capacity planning, throughput variance value which allows to define networking metrics more accurately especially for the throughput sensitive applications.

**Note**

Latency statistics are important for the latency critical applications like video streaming and VOIP traffic which usually do not tolerate latency variance [8]. For example, latency values of 250 ms are considered to be the maximum acceptable latency allowable in VOIP network.

Average value of the latency can be used for a baseline VOIP networking planning, while maximum value is more important. Latency variance may be an even more important parameter for the latency sensitive applications.

**Note**

Latency variation is usually called *Jitter* [4]

If jitter has a Gaussian distribution, it is usually quantified using the standard deviation of this distribution (aka. RMS). Often, jitter distribution is significantly non-Gaussian. This can occur if the jitter is caused by external sources. In these cases, peak-to-peak measurements are more useful. Many efforts have been made to meaningfully quantify distributions that are neither Gaussian nor have meaningful peaks (which is the case in all real jitter). More information about jitter measurement can be found in IETF RFC3393 [12].

VOIP applications usually have jitter buffers to collect and synchronize packet flow. Jitter value can be used to properly configure jitter buffers for such applications.

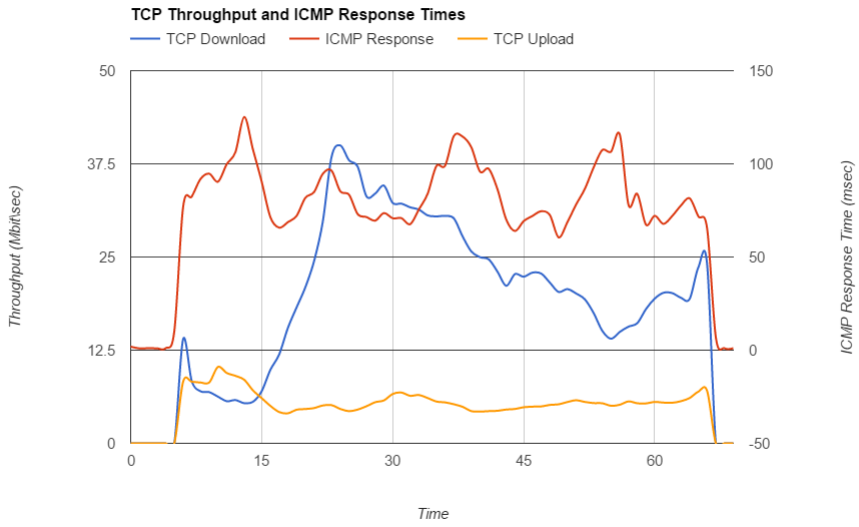



Figure 4.7: Combined throughput and latency graphs for individual worker

As discussed, an individual worker performance can provide significant information for the application architect and cloud operator who has to meet certain formal SLA or specific expectations from cloud users who host their applications on the cloud. Another important measurement is a dependency of the overall workers throughput versus number of workers. Actual behavior of this metric depends on the underlying physical networking architecture. If underlying network implements Fat Tree [2] DCN network topology the over-subscription will be 1:1 so that the whole networking bandwidth is available. If network over-subscription is different from 1:1, the behavior will depend on the workload distribution over physical infrastructure like nodes and racks. The example average throughput charts are presented in Figure 4.8,4.9 and 4.10

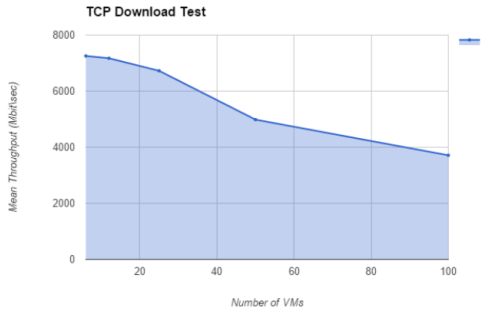



Figure 4.8: Average throughput vs number of VM workers in L2 test

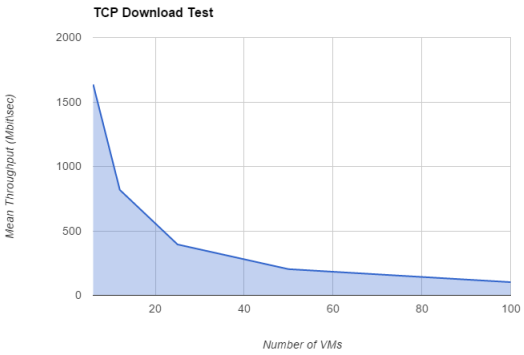



Figure 4.9: Average download throughput vs number of VM workers in L3 East-West test

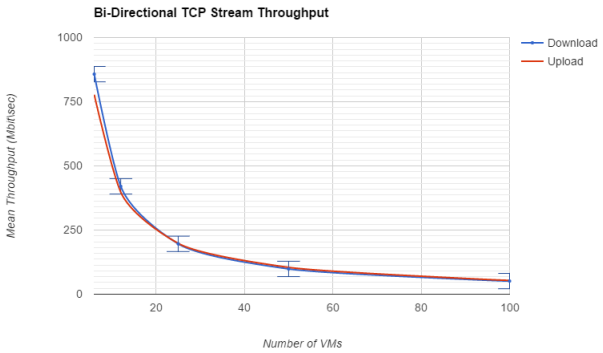
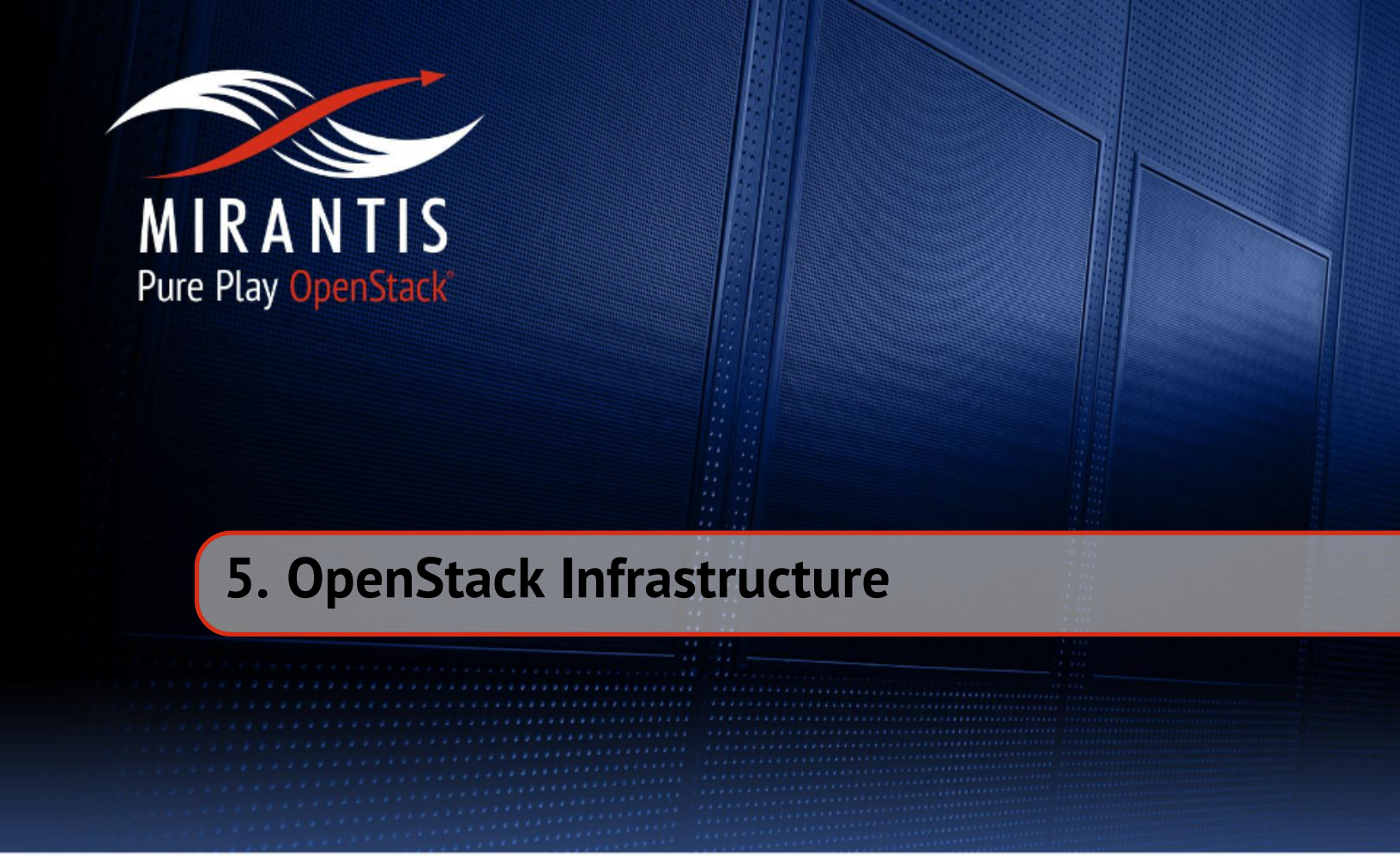



Figure 4.10: Average Upload and Download throughputs vs number of VM workers in L3 East-West test




# 5. OpenStack Infrastructure

OpenStack is a classical example of the Service Oriented Architecture. It uses one of the well known patterns – Message Bus, which is used when it is necessary to connect multiple independent applications or services [10]. Adapters and Common command structure components are implemented as an OpenStack library called *oslo.messaging* . Being a central part of the OpenStack services integration, the message bus system plays an important role in the overall OpenStack functionality as well as significantly affects the performance of a cloud. Performance and scalability of the messaging subsystem will be discussed further in this chapter in section 5.0.1. Another important part of the shared infrastructure in the OpenStack cloud is a database. OpenStack services are designed to be stateless services so that they can be scaled horizontally by adding new service instances if necessary. OpenStack services keep their state in the underlying DataBase layer. Being a shared resource, the DB layer also affects the overall performance and scalability of the entire cloud. Database performance and scalability testing is discussed in section 5.0.4.

### 5.0.1 Message Bus

OpenStack *oslo.messaging* library provides an abstraction layer for the messaging subsystem. It allows doing two different message communication patterns: RPC over MQ and notification publisher subscriber model. Following the Message Bus architecture pattern, the library provides all the necessary components for data model abstraction and translation as well as adapters for different underlying transport technologies. There are the following major components inside *oslo.messaging* :

**Transport** an abstraction layer to manage the underlying transport layer. There are sev-

eral adapters/drivers for different MQ implementations: (RabbitMQ, QPid, AMQP1, Kombu, ZeroMQ).

**Executors** Executors provide the way an incoming message will be dispatched so that the message can be used for meaningful work. Different types of executors are supported, each with its own set of restrictions and capabilities. Executor types are discussed below.

**Target** A Target encapsulates all the information to identify where a message should be sent or what messages a server is listening for.

**Server** An RPC server exposes a number of endpoints, each of which contains a set of methods which may be invoked remotely by clients over a given transport.

**RPC Client** The RPCClient class is responsible for sending method invocations to remote servers via a messaging transport.

**Notifier** The Notifier class is used for sending notification messages over a messaging transport or other means.

**Notification Listener** A notification listener exposes a number of endpoints, each of which contains a set of methods. Each method corresponds to a notification priority.

**Serializer** Generic (de-)serialization definition base class. Translates JSON messaging format to the Python objects.

### 5.0.2 Transport

RabbitMQ is a default driver used by *oslo.messaging* library. The same driver is aliased as *kombu* driver to support upgrading existing installations with older settings.

Qpid MQ driver is deprecated since Version 1.16 (Liberty).

AMQP1.0 driver is an experimental driver that provides support for Version 1.0 of the Advanced Message Queuing Protocol (AMQP 1.0, ISO/IEC 19464). The current implementation of this driver is considered experimental. It is not recommended that this driver be used in production systems. Rather, this driver is provided as a technical preview, in hopes that it will encourage further testing by the AMQP 1.0 community. This driver uses the Apache Qpid Proton AMQP 1.0 protocol engine. This engine consists of a platform specific library and a python binding. The driver does not directly interface with the engine API, as the API is a very low-level interface to the AMQP protocol. Instead, the driver uses the pure python Pyngus client API, which is layered on top of the protocol engine. The driver also requires a broker that supports Version 1.0 of the AMQP protocol. The driver has only been tested using *qpidd* in a patched devstack environment. The version of qpidd must be at least 0.26. *qpidd* also uses the Proton engine for its AMQP 1.0 support, so the Proton library must be installed on the system hosting the qpidd daemon.

At present, RabbitMQ does not work with this driver. This driver makes use of the dynamic flag on the link Source to automatically provision a node at the peer. RabbitMQ's AMQP 1.0 implementation has yet to implement this feature.

As for packages availability for different Linux distributives, the situation is the

following:

- RHEL and Fedora: Packages exist in EPEL for RHEL/CentOS 7, and Fedora 19+. Unfortunately, RHEL/CentOS 6 base packages include a very old version of qpidd that does not support AMQP 1.0. EPEL's policy does not allow a newer version of qpidd for RHEL/CentOS 6.
- Debian and Ubuntu: Packages for the Proton library, headers, and Python bindings are available in the Debian/Testing repository. Proton packages are not yet available in the Ubuntu repository. The version of qpidd on both platforms is too old and does not support AMQP 1.0. Until a proper package version arrives, the latest packages can be pulled from the Apache Qpid PPA on Launchpad.

Each transport layer behaves differently in the situation of multi-threading and process forking. *oslo.messaging* can't ensure that forking a process that shares the same transport object is safe for the library consumer, because it relies on different 3rd party libraries that do not ensure that. In certain cases, with some drivers, it does work:

- RabbitMQ: works only if no connection has already been established
- Qpid: does not work (The Qpid library has a global state that uses file descriptors that cannot be reset)
- amqp1: works

Multi-process usage of messaging should be tested by concurrency testing.

#### Executors

**aioeventlet** A message executor which integrates with eventlet and trollius. The executor is based on eventlet executor and is thus compatible with it. The executor supports trollius coroutines, explicit asynchronous programming, in addition to eventlet greenthreads, implicit asynchronous programming.

**blocking** A message executor which blocks the current thread.

The blocking executor's start() method functions as a request processing loop – i.e. it blocks, processes messages, and only returns when stop() is called from a dispatched method.

Method calls are dispatched in the current thread, so only a single method call can be executed at once. This executor is likely to only be useful for simple demo programs.

**eventlet** A message executor which integrates with eventlet.

This is an executor which polls for incoming messages from a greenthread and dispatches each message in its own greenthread powered async executor.

The stop() method kills the message polling greenthread and the wait() method waits for all executor maintained greenthreads to complete.

**threading** A message executor which integrates with threads.

A message process that polls for messages from a dispatching thread and on reception of an incoming message places the message to be processed in a thread pool to be executed at a later time.

**Target**

Different subsets of the information encapsulated in a Target object is relevant to various aspects of the API:

- creating a server – topic and server is required; exchange is optional
- an endpoint's target – namespace and version is optional
- client sending a message – topic is required, all other attributes optional

Target version numbers take the form Major.Minor. For a given message with version X.Y, the server must be marked as able to handle messages of version A.B, where A == X and B >= Y.

The Major version number should be incremented for an almost completely new API. The Minor version number would be incremented for backwards compatible changes to an existing API. A backwards compatible change could be something like adding a new method, adding an argument to an existing method (but not requiring it), or changing the type for an existing argument (but still handling the old type as well).

If no version is specified it defaults to '1.0'.

In the case of RPC, if you wish to allow your server interfaces to evolve such that clients do not need to be updated in lockstep with the server, you should take care to implement the server changes in backwards compatible and have the clients specify which interface version they require for each method.

Adding a new method to an endpoint is a backwards compatible change and the version attribute of the endpoint's target should be bumped from X.Y to X.Y+1. On the client side, the new RPC invocation should have a specific version specified to indicate the minimum API version that must be implemented for the method to be supported.

**Server**

An RPC server exposes a number of endpoints, each of which contains a set of methods which may be invoked remotely by clients over a given transport. The target supplied when creating an RPC server expresses the topic, server name and – optionally – the exchange to listen on. See Target for more details on these attributes.

Each endpoint object may have a target attribute which may have namespace and version fields set. By default, we use the 'null namespace' and Version 1.0. Incoming method calls will be dispatched to the first endpoint with the requested method, a matching namespace, and a compatible version number.

RPC servers have start(), stop() and wait() messages to begin handling requests, stop handling requests, and wait for all in-process requests to complete.

Clients can invoke methods on the server by sending the request to a topic and it gets sent to one of the servers listening on the topic, or by sending the request to a specific server listening on the topic, or by sending the request to all servers listening on the topic (known as fanout). These modes are chosen via the server and fanout attributes on

Target but the mode used is transparent to the server.

The first parameter to method invocations is always the request context supplied by the client.

Parameters to the method invocation are primitive types and so must be the return values from the methods. By supplying a serializer object, a server can deserialize a request context and arguments from – and serialize return values to – primitive types.

**Notification Listener** A notification listener exposes a number of endpoints, each of which contains a set of methods. Each method corresponds to a notification priority. The target supplied when creating a notification listener expresses the topic and – optionally – the exchange to listen on. See Target for more details on these attributes. Each notification listener is associated with an executor which integrates the listener with a specific I/O handling framework. Currently, there are blocking and eventlet executors available. A notifier sends a notification on a topic with a priority, the notification listener will receive this notification if the topic of this one has been set in one of the targets and if an endpoint implements the method named like the priority and if the notification match the NotificationFilter rule set into the filter_rule attribute of the endpoint.

Parameters to endpoint methods are the request context supplied by the client, the publisher_id of the notification message, the event_type, the payload and metadata. The metadata parameter is a mapping containing a unique message_id and a timestamp.

By supplying a serializer object, a listener can deserialize a request context and arguments from – and serialize return values to – primitive types.

By supplying a pool name, you can create multiple groups of listeners consuming notifications and that each group only receives one copy of each notification.

An endpoint method can explicitly return oslo_messaging.NotificationResult.HANDLED to acknowledge a message or oslo_messaging.NotificationResult.REQUEUE to re-queue the message.

The message is acknowledged only if all endpoints either return oslo_messaging.NotificationResult.HANDLED or None.

Note that not all transport drivers implement support for re-queueing. In order to use this feature, applications should assert that the feature is available by passing allow_requeue=True to get_notification_listener(). If the driver does not support re-queueing, it will raise NotImplementedError at this point.

### 5.0.3 MessageBus Testing

**Remark**

As RabbitMQ is a default MQ transport used by OpenStack, the test plan will refer to RabbitMQ here. The same tests are valid for other MQ implementations but will require the change of the parameter names.

There are a huge number of variables that feed into the overall level of performance you can get from a RabbitMQ server and it is important to know how each of them

contributes to the performance characteristics of the server. Test numbers usually correlate with the hardware specification, therefore, these numbers should be normalized to CPU/GHz and Memory/GB amount. Scalability tests should reveal the limits of the MQ capacity and reveal their relation to hardware resources. The theoretical relations is known as the number of queues is limited by size of memory available, the number of messages in process is limited by disk capacity. The number of subscribers can be limited by the number of TCP sockets limits in the system. A good RabbitMQ analysis is done in articles [30], [31].

**Measurements**

There exist different approaches on the performance characterization of the queues. The first one is purely theoretical an operates with:

- Traffic Intensity $\rho$
- Utilisation $U$
- Average number of messages in a queue $L_q$
- Average number of messages in a system $L_s$
- Arrival rate $\lambda$
- Average time spent in a queue $W_q$
- Average time spent in a system $W_s$
- Service processing rate $\mu$
- Service time $S = 1/\mu$

There is a whole theory around queues which has significant results around specific queue modeling and description of mathematical properties of such models. For a specific arrival statistical process (i.e. Poisson, Deterministic, general), there are specific relations between performance characteristics of the queue[34]. We already met with one of these relations known as Little's Law 5.1.

$$\begin{aligned} L_q &= \lambda W_q \\ L_s &= \lambda W_s \end{aligned} \tag{5.1}$$

Other important relations for Poisson distributed incoming process are the following:

$$\begin{aligned} W_q &= \frac{\rho}{\mu - \lambda} \\ W_q &= W - \frac{1}{\mu} \end{aligned} \tag{5.2}$$

$$L_q = \rho^2/(1-\rho), \rho = \frac{\lambda}{\mu} \tag{5.3}$$

$$U = \lambda S = \rho, S = \frac{1}{\mu} \tag{5.4}$$

Measuring two major parameters – arrival rate $\lambda$ and service rate $\mu$ it is possible to calculate all other performance characteristics using some assumptions about distribution model. For example, the following is correct for a simple queue with a single service and Poisson distributed arrival process. If a system receives messages with a rate of 240 messages per minute, and the message length is in average 176 bytes, while outgoing transmission rate is 800 bytes per second, the performance characteristics of the system are the following:

- *Arrival rate* $\lambda = 4msg/sec$
- *Service rate* $\mu = 800/176 = 4.54msg/sec$
- *Utilization* $\rho = \lambda/\mu = 0.88 = 88\%$
- *Average queue length* $L_q = \rho^2/(1-\rho) = 6.4msgs$
- *Average time spent in system* $W = 1.8sec$
- *Average time spent in a queue* $W_q = W - 1/\mu = 1.57sec$

The expected value and the variance of the minimum (maximum) number of customers in the system (queue) as well as the $n^{th}$ moments of the minimum (maximum) waiting time in the queue can be found in article [5].

Theoretical relations between different performance characteristics can be used for the indirect measurements as we discussed in section 3.1.2. There are several values which can be measured directly. Some of them are discussed in the work published by RabbitMQ team [20]. The measured values are:

- Sending message rate
- Receiving message rate
- Latency

The work [20] clearly shows that for the different RabbitMQ versions the queue performance behavior is different and is quite far from the theoretical models. This means that all these theoretical formulas, which are usually used in capacity planning, should be used only as a reference, and the actual system performance should be measured for better accuracy.

#### Measurement Methodology

The MQ measurement process should use direct measured metrics which were discussed in the previous section. Most of the MQ monitoring services as well as MQ internal stats provide sufficient information for the direct measurement.

**Note**

RabbitMQ uses disk space to store messages inside the service. Warm-up period might be necessary to have reliable performance results.

In addition to basic rate values, some of the MQ can provide stats about:

- Number of queues
- TPS for each queue
- Number of messages in a queue

These statistics can be gathered and used in the MQ performance analysis. There are several tools available for MQ performance testing:

- OpenStack oslo.messaging simulator [26]
- RabbitMQ Performance Tool [27]
- ActiveMQ JMeter plugin [1]

Measurement methodology can be different for different aspects of performance testing. For MQ systems, it is possible to specify the following aspects of performance testing:

**Connection Load** The number of message producers, or message consumers, or the number of concurrent connections a system can support.

**Message throughput** The number of messages or message bytes that can be pumped through a messaging system per second.

**Latency** The time it takes a particular message to be delivered from message producer to message consumer.

**Stability** The overall availability of the message service or how gracefully it degrades in cases of heavy load or failure.

**Efficiency** The efficiency of message delivery: a measure of message throughput in relation to the computing resources employed.

These different aspects of performance are generally inter-related. If message throughput is high, this means that the messages are less likely to be backlogged in the message server, and, as a result, latency should be low (a single message can be delivered very quickly). However, latency can depend on many factors: the speed of communication links, message server processing speed, and client processing speed, to name a few.

In any case, there are several different aspects of performance. Which of them are the most important to you generally depends on the requirements of a particular application.

**Important**

In order to have reliable results for a specific usage, it is necessary to establish baseline use patterns. It should include information about peak demand occurrence, fluctuations in the number of users, number of messages, and typical producing/consuming rates. To establish baseline use patterns, it is possible to collect monitoring statistics from the actual system over an extended period of time. This data should include but not be limited to:

- Number of concurrent connections (active and not used)
- Number of messages stored in the broker. Per queue or overall.
- Incoming and outgoing rates. Per queue or aggregated.
- Number of active consumers.

Once the baseline usage patterns are established, they can be used in the measurement process. The following queue processing metrics should be collected for the MQ system. Each measurement should be analyzed with the approach used in the API testing. The list of collected values is presented in the table 5.1

**Important**

- **Stationarity**. Verify that MQ subsystem behavior is consistent over a large time frame. If

| Usage Pattern | $\lambda$ (msg/sec) | $\mu$(msg/sec) | $\rho$ | $L_q$ (msgs) | $W_q$ (sec) |
|---|---|---|---|---|---|
| Steady Load | | | | | |
| Burst Load | | | | | |
| PatternX Load | | | | | |
| ... | | | | | |

Table 5.1

the process is stationary, it is possible to use measured values confidently independent of the system load time.

- **Distribution type**. If the measured values follow the standard distribution, the mean value can be used as an actual measured value.
- In case of non-standard distribution, additional analysis should be performed to understand the type of underlying system dynamics. If it is possible to introduce several character times (clustering analysis), each value should be clearly attributed to the specific process and provided with applicability restrictions (i.e. specific usage pattern or specific message size).

Message rates $\lambda$ and $\mu$ can be directly measured on the client sides or on the networking level. $L_q$ and $W_q$ might be available from the MQ statistics, and logs and $W_q$ could be directly measured via latency measurement.

In addition to queue specific measurements, it is necessary to do CPU and memory consumption testing to be able do capacity analysis. RabbitMQ uses disk to store message content, so disk subsystem performance should be tested as well. RabbitMQ memory consumption does not depend (to an extent) on the number of messages but depends on the number of queues and connections [30].

The following graph should be measured:

- CPU usage vs. concurrent connections
- Memory usage vs. concurrent connections
- CPU usage vs. incoming rate
- Memory usage vs. incoming rate
- Disk usage vs. message size vs. number of queues
- CPU usage vs. number of queues
- Memory usage vs. number of queues

Message queue performance can be affected by the actual usage patterns and options (factors) used by MQ clients. There are several key factors which might affect the MQ performance:

**Delivery Mode** Persistent messages guarantee message delivery in case of message server failure. The broker stores the message in a persistent store until all intended consumers acknowledge they have consumed the message. The difference in performance between the persistent and non-persistent modes can be significant.

**Use of Transactions** A transaction is a guarantee that all messages produced in a trans-

acted session and all messages consumed in a transacted session will be either processed or not processed (rolled back) as a unit.

**Acknowledgement Mode** One mechanism for ensuring the reliability of MQ message delivery is for a client to acknowledge consumption of messages delivered to it by the Message Queue message server.

**Durable and Non-durable Subscriptions** Subscribers to a topic destination fall into two categories, those with durable and non-durable subscriptions. Durable subscriptions provide increased reliability but slower throughput.

**Use of Exchanges and Routing Keys (Message Filtering)** Usage of routing keys and exchanges may affect the performance as it requires more processing of the message on the MQ side.

**Message Size** Message size affects performance because more data must be passed from producing client to broker and from broker to consuming client, and because for persistent messages a larger message must be stored.

Table 5.2 provides a comparison of the factors and their applicability:

| MQ Usage Factor | High Reliability/Low Performance | High Performance/Low Reliability |
|---|---|---|
| Delivery mode | Persistent messages | Non-persistent messages |
| Use of transactions | Transacted sessions | No transactions |
| Acknowledgment mode | AUTO_ACKNOWLEDGE or CLIENT_ACKNOWLEDGE | DUPS_OK_ACKNOWLEDGE |
| Durable/non-durable subscriptions | Durable subscriptions | Non-durable subscriptions |
| Use of selectors | Message filtering | No message filtering |
| Message size | Large number of small messages | Small number of large messages |

Table 5.2

Typical items in an MQ test plan case are the following:

- Define MQ Server URL
- Define test duration (min)
- Define Ramp Up (Warm Up) period (min)
- Define number of producers
- Define number of consumers/subscribers
- Define message size (or size distribution)
- Define queue/exchange
- Define topic is filtering is used
- Define Delivery Mode

### 5.0.4 Database


# 6. Lab and Testing Tools

### 6.0.1 Control Plane (API) Testing Tools

**JMeter**

The leader of the pack in awareness is probably Apache JMeter. This is an open-source Java application whose key feature is a powerful and complete GUI which you use to create test plans. A test plan is composed of test components which define every piece of the test such as:

- Multiple threads to generate a load
- Parametrizing HTTP requests
- Flexible output results via listeneres interface

This tool is very well established and is probbaly one of the best tools for functional load testing. It allows to model complex flows using conditions and allows to create custom assertions to validate the behavior. It also allows to simulate non-trivial HTTP scenarios like logging in before the actual HTTP call to a specific URL or perform file uploads. JMeter has a wide and well established community which produces various plugins to modify and extend the built-in behaviors. JMeter allows to test not only HTTP based API but also supports various protocols including:

- Web - HTTP and HTTPS
- SOAP and REST API protocols (over HTTP)
- FTP
- Databases via JDBC Java DB interfaces
- LDAP
- Message oriented middleware MOM via JMS
- Mail - SMTP, POP3 and IMAP

- MongoDB
- TCP over IP

And last, but not the least, JMeter is open source and free. As with every tool, it has its own limitations and problems. JMeter comes with GUI which has a steep learning curve. It is overloaded with options and concepts which one should learn before being able to use this tool efficiently. GUI consumes a lot of compute and memory resources, so, in order to reduce the performance impact, the GUI can be switched off and tests can be executed in non-GUI mode. It will require saving the test scenario in an XML formatted file and using CLI tool to start the test execution. The desired throughput of the requests is controlled by several parameters of the test scenario and requires fine-tuning before test execution.

#### Gatling

Gatling is a highly capable load testing tool. It is designed for ease of use, maintainability, and high performance.

Out of the box, Gatling comes with excellent support of the HTTP protocol that makes it a tool of choice for load testing of any HTTP server. As the core engine is actually protocol agnostic, it is perfectly possible to implement support for other protocols. For example, Gatling currently also ships JMS support. Gatling's architecture is asynchronous as long as the underlying protocol, such as HTTP, can be implemented in a non-blocking way. This kind of architecture lets us implement virtual users as messages instead of dedicated threads, making them very resource cheap. Thus, running thousands of concurrent virtual users is not an issue. Gatling uses its own DSL for the test scenarios.

#### Wrk and Apache AB

Wrk and Apache AB are command line tools to test HTTP based resources. In these tools everything is configured via command line interface through command line parameters. It has few powerful setting essential to generate HTTP load. As a result of simplicity both tools are capable to generate high loads. It can be also extended via plugins. There are plugins for Kafka and RabbitMQ tests.

#### Rally

Rally is a benchmarking tool that was designed specifically for OpenStack API testing. To make this possible, Rally automates and unifies multi-node OpenStack deployment, cloud verification, benchmarking & profiling. Rally does it in a generic way, making it possible to check whether OpenStack is going to work well on, say, a 1k-servers installation under high load. The actual Rally core consists of four main components, listed below in the order they go into action:

**Server Providers** provide a unified interface for interaction with different virtualization technologies (LXS, Virsh etc.) and cloud suppliers (like Amazon): it does so via SSH access and in one L3 network

**Deploy Engines** deploy some OpenStack distribution (like DevStack or Fuel) before any benchmarking procedures take place, using servers retrieved from Server Providers

**Verification** runs Tempest (or another specific set of tests) against the deployed cloud to check that it works correctly, collects results and presents them in a human readable form

**Benchmark Engine** allows to write parameterized benchmark scenarios & run them against the cloud

Rally is written in Python language and can easily be extended with plugins written in Python. Types of Rally plugins are presented on picture 6.1.

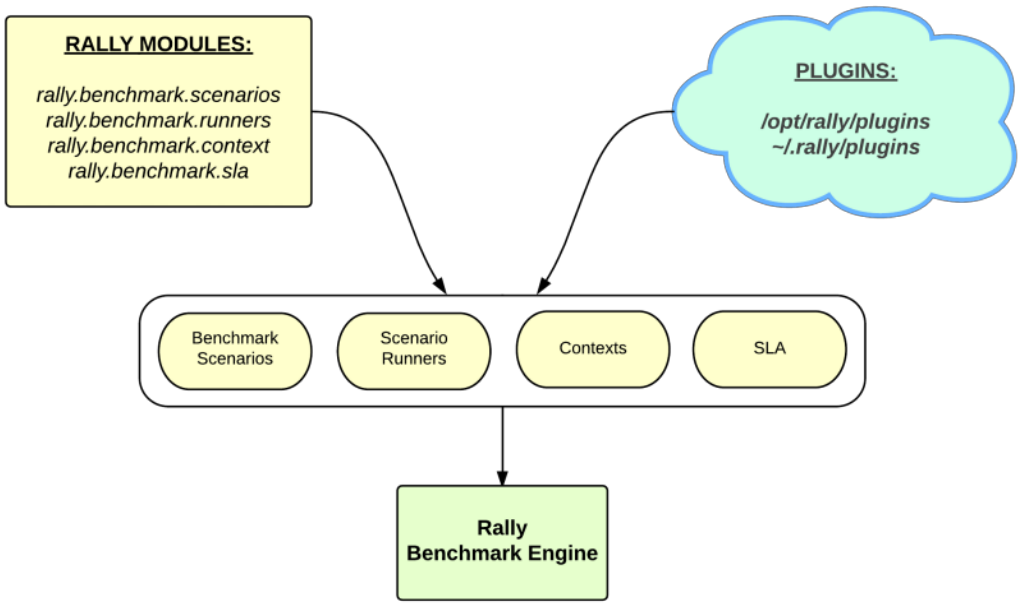



Figure 6.1: Rally pluggable architecture

### 6.0.2 Data Plane Testing Tools

#### Shaker

The **Shaker** tool is a tool used and developed by Mirantis to understand the Data Plane capabilities of an OpenStack deployment. Data Plane testing helps cloud administrators understand their deployment from the perspective of the applications that are using the environment. This tool can be used for deployment planning, environment verification, and troubleshooting.

Today, Shaker focuses on network based tests using *iperf* to drive load across the network. Shaker has future plans to roll out testing to evaluate I/O and CPU.

Shaker utilizes Heat templates to deploy and execute Data Plane tests. It deploys a number of agents/compute nodes that all report back to a centralized Shaker server (Fig.6.2).

The server is executed by *shaker* command and is responsible for deployment of instances, execution of tests as specified in the scenario, for results processing and report generation. The agent is light-weight and polls tasks from the server and replies with

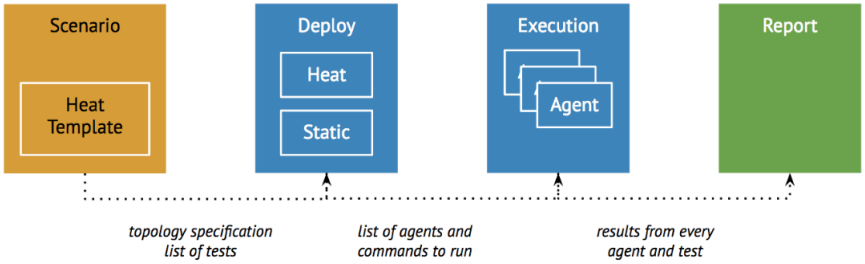



Figure 6.2: Shaker test processing workflow

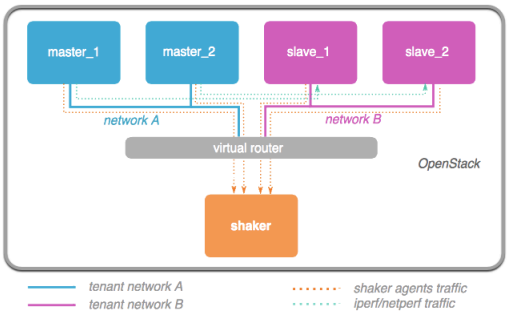



Figure 6.3: Shaker architecture

the results. Agents have connectivity to the server, but the server does not (so it is easy to keep agents behind NAT) (Fig.6.3).

### 6.0.3 Logging, Monitoring and Alerting Toolchain

The Mirantis OpenStack LMA (Logging, Monitoring and Alerting) Toolchain is comprised of a collection of open-source tools to simpify monitoring and issues diagnosis in the OpenStack environments. These tools are packaged and delivered as Mirantis Fuel plugins that can be installed using the graphic user interface of Mirantis Fuel starting with Mirantis OpenStack version 6.1.

From a high level view, the LMA Toolchain includes:

**LMA Collectors** gather relevant operational data to increase the operational visibility over the OpenStack environment. Those data are collected from a variety of sources including the log messages from cloud nodes, collectd, and the OpenStack notifications bus. Collectors can be described as a pluggable message processing and routing pipeline. Its components are:

- Collectd that is bundled with a monitoring plugins collection.
- Heka - keystone component of the Collector.
- Collection of Heka plugins written in Lua to decode, process and encode the data to be sent to the external systems.

**Satellite Clusters** are pluggable external systems which can take an action on the data

received from the Collectors running on the OpenStack nodes. Curretly Satellite Clusters built-in to the LMA Toolchain are:

- Elasticsearch, powerful open source search server that makes data like log messages and notifications easy to explore and analyse. Kibana Dashboar is used to simplify Elasticsearch usage and logs analysis.
- InfluxDB, open-source and distributed time series database to store and analyse metrics collected. Open source Grafana Dashboard is used to visualize time series data and to show how does OpenStack behaves over time.

It is quite possible to integrate the Collector with an external monitoring applications like Nagios. This could simply be done through enabling the Nagios output plugin of Heka for a subset of messages matching the message matcher syntax of the output plugin.

**Log Messages**

The Heka collector service is configured to tail the following log files:

- System logs:
  - /var/log/syslog
  - /var/log/messages
  - /var/log/debug
  - /var/log/auth.log
  - /var/log/cron.log
  - /var/log/daemon.log
  - /var/log/kern.log
  - /var/log/pacemaker.log
- MySQL server logs (for controller nodes).
- RabbitMQ server logs (for controller nodes).
- Pacemaker logs (for controller nodes).
- OpenStack services logs.
- Open vSwitch logs (all nodes):
  - /var/log/openvswitch/ovsdb-server.log
  - /var/log/openvswitch/ovs-vswitchd.log

**Notification Messages**

OpenStack services can be configured to send notifications on the message bus about the task execution or the cloud resources state change. These notifications are received by the LMA Collector service and turned into the Heka messages.

**Metric Messages**

Metrics are extracted from the data received from collectd, log messages processed by the Collector service and OpenStack notifications processed by the Collector service.

The list of collected metrics is presented in sections below.

**System**

- CPU (<cpu number> expands to 0, 1, 2, and so on)

  - cpu.<cpu number>.idle - percentage of CPU time spent in the idle task.
  - cpu.<cpu number>.interrupt - percentage of CPU time spent servicing interrupts
  - cpu.<cpu number>.nice - percentage of CPU time spent in user mode with low priority (nice).
  - cpu.<cpu number>.softirq - percentage of CPU time spent servicing soft interrupts.
  - cpu.<cpu number>.steal - percentage of CPU time spent in other operating systems.
  - cpu.<cpu number>.system - percentage of CPU time spent in system mode.
  - cpu.<cpu numbner>.user - percentage of CPU time spent in user mode.
  - cpu.<cpu number>.wait - percentage of CPU time spent waiting for I/O operations to complete.
  -
- Disk (<disk device> expands to 'sda', 'sdb' and so on)
  - disk.<disk device>.disk_merged.read - the number of read operations per second that could be merged with already queued operations.
  - disk.<disk device>.disk_merged.write - the number of write operations per second that could be merged with already queued operations.
  - disk.<disk device>.disk_octets.read - the number of octets (bytes) read per second.
  - disk.<disk device>.disk_octets.write - the number of octets (bytes) written per second.
  - disk.<disk device>.disk_ops.read - the number of read operations per second.
  - disk.<disk device>.disk_ops.write - the number of write operations per second.
  - disk.<disk device>.disk_time.read - the average time for a read operation to complete in the last interval.
  - disk.<disk device>.disk_time.write - the average time for a write operation to complete in the last interval.
- File System (<mount point> expands to 'root' for '/', 'boot' for '/boot', 'var-lib' for '/var/lib' and so on)
  - fs.<mount point>.inodes.free - the number of free inodes on the file system.
  - fs.<mount point>.inodes.reserved - the number of reserved inodes.
  - fs.<mount point>.inodes.used - the number of used inodes.
  - fs.<mount point>.space.free - the number of free bytes.
  - fs.<mount point>.space.reserved - the number of reserved bytes.
  - fs.<mount point>.space.used - the number of used bytes.
- System Load
  - load.longterm - the system load average over the last 15 minutes.
  - load.midterm - the system load average over the last 5 minutes.
  - load.shortterm - the system load averge over the last minute.
- Memory
  - memory.buffered - the amount of memory (in bytes) which is buffered.

  - memory.cached - the amount of memory (in bytes) which is cached.
  - memory.free - the amount of memory (in bytes) which is free.
  - memory.used - the amount of memory (in bytes) which is used.
- Network (<interface> expands to the interface name, eg 'br-mgmt', 'br-storage' and so on)
  - net.<interface>.if_errors.rx - the number of errors per second detected when receiving from the interface.
  - net.<interface>.if_errors.tx - the number of errors per second detected when transmitting from the interace.
  - net.<interface>.if_octets.rx - the number of octets (bytes) received per second by the interace.
  - net.<interface>.if_octets.tx - the number of octets (bytes) transmitted per second by the interface.
  - net.<interface>.if_packets.rx - the number of packets received per second by the interace.
  - net.<interface>.if_packets.tx - the number of packets transmitted per second by the interface.
- Processes
  - processes.fork_rate - the number of processes forked per second.
  - processes.state.blocked - the number of processes in blocked state.
  - processes.state.paging - the number of processes in paging state.
  - processes.state.running - the number of processes in running state.
  - processes.state.sleeping - the number of processes in sleeping state.
  - processes.state.stopped - the number of processes in stopped state.
  - processes.state.zombies- the number of processes in zombie state.
- Swap
  - swap.cached - the amount of cached memory (in bytes) which is in the swap.
  - swap.free - the amount of free memory (in bytes) which is in the swap.
  - swap.used - the amount of used memory (in bytes) which is in the swap.
  - swap_io.in - the number of swap pages written per second.
  - swap_io.out - the number of swap pages read per second.
- Users
  - users - the number of users currently logged-in.

**Apache**

- apache.status - the status of the Apache service, 1 if it is responsive, 0 otherwise.
- apache.bytes - the number of bytes per second transmitted by the server.
- apache.requests - the number of requests processed per second.
- apache.connections - the current number of active connections.
- apache.idle_workers - the current number of idle workers.
- apache.workers.<state> - the current number of workers by state (<state> is one of closing, dnslookup, finishing, idle_cleanup, keepalive, logging, open, reading, sending, starting, waiting).

**MySQL**

- Service
  - mysql - the status of the MySQL service, 1 if it is responsive, 0 otherwise.
- Commands
  - mysql_commands.admin_commands - the number of ADMIN statements.
  - mysql_commands.change_db - the number of USE statements.
  - mysql_commands.commit - the number of COMMIT statements.
  - mysql_commands.flush - the number of FLUSH statements.
  - mysql_commands.insert - the number of INSERT statements.
  - mysql_commands.rollback - the number of ROLLBACK statements.
  - mysql_commands.select - the number of SELECT statements.
  - mysql_commands.set_option - the number of SET statements.
  - mysql_commands.show_collations - the number of SHOW COLLATION statements.
  - mysql_commands.show_databases - the number of SHOW DATABASES statements.
  - mysql_commands.show_fields - the number of SHOW FIELDS statements.
  - mysql_commands.show_master_status - the number of SHOW MASTER STATUS statements.
  - mysql_commands.show_status - the number of SHOW STATUS statements.
  - mysql_commands.show_tables - the number of SHOW TABLES statements.
  - mysql_commands.show_variables - the number of SHOW VARIABLES statements.
  - mysql_commands.show_warnings - the number of SHOW WARNINGS statements.
  - mysql_commands.update - the number of UPDATE statements.
- Handlers
  - mysql_handler.commit - the number of internal COMMIT statements.
  - mysql_handler.delete - the number of internal DELETE statements.
  - mysql_handler.external_lock - the number of external locks.
  - mysql_handler.read_first - the number of times the first entry in an index was read.
  - mysql_handler.read_key - the number of requests to read a row based on a key.
  - mysql_handler.read_next - the number of requests to read the next row in key order.
  - mysql_handler.read_prev - the number of requests to read the previous row in key order.
  - mysql_handler.read_rnd - the number of requests to read a row based on a fixed position.
  - mysql_handler.read_rnd_next - the number of requests to read the next row in the data file.
  - mysql_handler.rollback - the number of requests for a storage engine to

perform rollback operation.
  - mysql_handler.update - the number of requests to update a row in a table.
  - mysql_handler.write - the number of requests to insert a row in a table.
- Locks
  - mysql_locks.immediate - the number of times per second the requests for table locks could be granted immediately.
  - mysql_locks.waited - the number of times per second the requests for table locks had to wait.
- Network
  - mysql_octets.rx - the number of bytes received per second by the server.
  - mysql_octets.tx - the number of bytes sent per second by the server.
- Query Cache
  - mysql_qcache.hits - the number of query cache hits per second.
  - mysql_qcache.inserts - the number of queries added to the query cache per second.
  - mysql_qcache.lowmem_prunes - the number of queries that were deleted from the query cache per second because of low memory.
  - mysql_qcache.not_cached - the number of noncached queries per second.
  - mysql_qcache.queries_in_cache - the number of queries registered in the query cache per second.
- Threads
  - mysql_threads.cached - the number of threads in the thread cache.
  - mysql_threads.connected - the number of currently open connections.
  - mysql_threads.running - the number of threads that are not sleeping.
  - mysql_threads.created - the number of threads created per second to handle connections.
- Cluster
  - mysql.cluster.size - current number of nodes in the cluster.
  - mysql.cluster.status - 1 when the node is 'Primary', 2 if 'Non-Primary' and 3 if 'Disconnected'.
  - mysql.cluster.connected - 1 when the node is connected to the cluster, 0 otherwise.
  - mysql.cluster.ready - 1 when the node is ready to accept queries, 0 otherwise.
  - mysql.cluster.local_commits - number of writesets commited on the node.
  - mysql.cluster.received_bytes - total size in bytes of writesets received from other nodes.
  - mysql.cluster.received - total number of writesets received from other nodes.
  - mysql.cluster.replicated_bytes - total size in bytes of writesets sent to other nodes.
  - mysql.cluster.replicated - total number of writesets sent to other nodes.
  - mysql.cluster.local_cert_failures - number of writesets that failed the certification test.
  - mysql.cluster.local_send_queue - the number of writesets waiting to be sent.

  - mysql.cluster.local_recv_queue - the number of writesets waiting to be applied.
- Slow Queries
  - mysql.slow_queries - number of queries that have taken more than X seconds, depending of the MySQL configuration parameter 'long_query_time' (10s per default).

**RabbitMQ**

- Service
  - rabbitmq.status - the status of the RabbitMQ service, 1 if it is responsive, 0 otherwise.
- Cluster
  - rabbitmq.connections - Number of connections.
  - rabbitmq.consumers - number of consumers.
  - rabbitmq.exchanges - number of exchanges.
  - rabbitmq.memory - bytes of memory consumed by the Erlang process associated with all queues, including stack, heap and internal structures.
  - rabbitmq.messages - total number of messages which are ready to be consumed or not yet acknowledged.
  - rabbitmq.total_nodes - number of nodes in the cluster.
  - rabbitmq.running_nodes - number of running nodes in the cluster.
  - rabbitmq.queues - number of queues.
- Queries
  - rabbitmq.<name_of_the_queue>.consumers - number of consumers.
  - rabbitmq.<name_of_the_queue>.memory - bytes of memory consumed by the Erlang process associated with the queue, including stack, heap and internal structures.
  - rabbitmq.<name_of_the_queue>.messages - number of messages which are ready to be consumed or not yet acknowledged.

**HAProxy**

Frontends and backends used further are one of the:

- cinder-api
- glance-api
- glance-registry-api
- heat-api
- heat-cfn-api
- heat-cloudwatch-api
- horizon-web
- keystone-public-api
- keystone-admin-api
- mysqld-tcp
- murano-api
- neutron-api

- nova-api
- nova-ec2-api
- nova-metadata-api
- nova-novncproxy-websocket
- sahara-api
- swift-api
- Server
  - haproxy.status - the status of the HAProxy service, 1 if it is responsive, 0 otherwise.
  - haproxy.connections - number of current connections.
  - haproxy.ssl_connections - number of current SSL connections.
  - haproxy.pipes_free - number of free pipes.
  - haproxy.pipes_used - number of used pipes.
  - haproxy.run_queue - number of connections waiting in the queue.
  - haproxy.tasks - number of tasks.
  - haproxy.uptime - HAProxy server uptime in seconds.
- Frontends
  - haproxy.frontend.<frontend>.bytes_in - number of bytes received by the frontend.
  - haproxy.frontend.<frontend>.bytes_out - number of bytes transmitted by the frontend.
  - haproxy.frontend.<frontend>.denied_requests - number of denied requests.
  - haproxy.frontend.<frontend>.denied_responses - number of denied responses.
  - haproxy.frontend.<frontend>.error_requests - number of error requests.
  - haproxy.frontend.<frontend>.response_1xx - number of HTTP responses with 1xx code.
  - haproxy.frontend.<frontend>.response_2xx - number of HTTP responses with 2xx code.
  - haproxy.frontend.<frontend>.response_3xx - number of HTTP responses with 3xx code.
  - haproxy.frontend.<frontend>.response_4xx - number of HTTP responses with 4xx code.
  - haproxy.frontend.<frontend>.response_5xx - number of HTTP responses with 5xx code.
  - haproxy.frontend.<frontend>.response_other - number of HTTP responses with other code.
  - haproxy.frontend.<frontend>.session_current - number of current sessions.
  - haproxy.frontend.<frontend>.session_total - cumulative of total number of session.
  - haproxy.frontend.bytes_in - total number of bytes received by all frontends.
  - haproxy.frontend.bytes_out - total number of bytes transmitted by all frontends.
  - haproxy.frontend.session_current - total number of current sessions for all

frontends.

- Backends
  - haproxy.backend.<backend>.bytes_in - number of bytes received by the backend.
  - haproxy.backend.<backend>.bytes_out - number of bytes transmitted by the backend.
  - haproxy.backend.<backend>.denied_requests - number of denied requests.
  - haproxy.backend.<backend>.denied_responses - number of denied responses.
  - haproxy.backend.<backend>.downtime - total downtime in second.
  - haproxy.backend.<backend>.status - the backend status where values 0 and 1 represent respectively DOWN and UP.
  - haproxy.backend.<backend>.error_connection - number of error connections.
  - haproxy.backend.<backend>.error_responses - number of error responses.
  - haproxy.backend.<backend>.queue_current - number of requests in queue.
  - haproxy.backend.<backend>.redistributed - number of times a request was redispatched to another server.
  - haproxy.backend.<backend>.response_1xx - number of HTTP responses with 1xx code.
  - haproxy.backend.<backend>.response_2xx - number of HTTP responses with 2xx code.
  - haproxy.backend.<backend>.response_3xx - number of HTTP responses with 3xx code.
  - haproxy.backend.<backend>.response_4xx - number of HTTP responses with 4xx code.
  - haproxy.backend.<backend>.response_5xx - number of HTTP responses with 5xx code.
  - haproxy.backend.<backend>.response_other - number of HTTP responses with other code.
  - haproxy.backend.<backend>.retries - number of times a connection to a server was retried.
  - haproxy.backend.<backend>.servers.down - number of servers which are down.
  - haproxy.backend.<backend>.servers.up - number of servers which are up.
  - haproxy.backend.<backend>.session_current - number of current sessions.
  - haproxy.backend.<backend>.session_total - cumulative number of sessions.
  - haproxy.backend.bytes_in - total number of bytes received by all backends.
  - haproxy.backend.bytes_out - total number of bytes transmitted by all backends.
  - haproxy.backend.queue_current - total number of requests in queue for all backends.
  - haproxy.backend.session_current - total number of current sessions for all backends.
  - haproxy.backend.error_responses - total number of error responses for all backends.

**Memcached**

- memcached.status - the status of the memcached service, 1 if it is responsive, 0 otherwise.
- memcached.command.flush - cumulative number of flush reqs.
- memcached.command.get - cumulative number of retrieval reqs.
- memcached.command.set - cumulative number of storage reqs.
- memcached.command.touch - cumulative number of touch reqs.
- memcached.connections.current - number of open connections.
- memcached.items.current - current number of items stored.
- memcached.octets.rx - total number of bytes read by this server from network.
- memcached.octets.tx - total number of bytes sent by this server to network.
- memcached.ops.decr_hits - number of successful decr reqs.
- memcached.ops.decr_misses - number of decr reqs against missing keys.
- memcached.ops.evictions - number of valid items removed from cache to free memory for new items.
- memcached.ops.hits - number of keys that have been requested.
- memcached.ops.incr_hits - number of successful incr reqs.
- memcached.ops.incr_misses - number of successful incr reqs.
- memcached.ops.misses - number of items that have been requested and not found.
- memcached.df.cache.used - current number of bytes used to store items.
- memcached.df.cache.free - current number of free bytes to store items.
- memcached.percent.hitratio - percentage of get command hits (in cache).

**OpenStack**

- Service Checks
  - openstack.<service>.check_api] the service's API status, 1 if it is responsive, 0 otherwise. <service> is one of the following services with their respective resource checks:
    - 'nova': '/'
    - 'cinder': '/'
    - 'cinder-v2': '/'
    - 'glance': '/'
    - 'heat': '/'
    - 'keystone': '/'
    - 'neutron': '/'
    - 'ceilometer': '/v2/capabilities'
    - 'swift': '/healthcheck'
    - 'swift-s3': '/healthcheck'
- Compute (<state> is one of 'active', 'deleted', 'error', 'paused', 'resumed', 'rescued', 'resized', 'shelved_offloaded' or 'suspended', <service> is one of service is one of 'compute', 'conductor', 'scheduler', 'cert' or 'consoleauth', <service_state> is one of 'up', 'down' or 'disabled')
  - openstack.nova.instance_creation_time - the time (in seconds) it took to launch

a new instance.
  - openstack.nova.instance_state.<state> - the number of instances which entered this state.
  - openstack.nova.total_free_disk - the total amount of disk space (in GB) available for new instances.
  - openstack.nova.total_used_disk - the total amount of disk space (in GB) used by the instances.
  - openstack.nova.total_free_ram - the total amount of memory (in MB) available for new instances.
  - openstack.nova.total_used_ram - the total amount of memory (in MB) used by the instances.
  - openstack.nova.total_free_vcpus - the total number of virtual CPU available for new instances.
  - openstack.nova.total_used_vcpus - the total number of virtual CPU used by the instances.
  - openstack.nova.total_running_instances - the total number of running instances.
  - openstack.nova.total_running_tasks - the total number of tasks currently executed.
  - openstack.nova.instances.<state> - the number of instances by state.
  - openstack.nova.services.<service>.<service_state> - the total number of Nova services by state.
  - openstack.nova.services.<service>.status - status of Nova services computed from openstack.nova.services.<service>.<service_state>.
  - openstack.nova.api.<backend>.status - status of the API services located behind the HAProxy load-balancer, computed from haproxy.backend.nova-*.servers.(up|down).
  - openstack.nova.status - the general status of the Nova service which is computed using the previous metrics and the openstack.nova.check_api metric.
- Identity (<state> is one of 'disabled' or 'enabled')
  - openstack.keystone.roles - the total number of roles.
  - openstack.keystone.tenants.<state> - the number of tenants by state.
  - openstack.keystone.users.<state> - the number of users by state.
  - openstack.keystone.api.<backend>.status - status of the API services located behind the HAProxy load-balancer, computed from haproxy.backend.keystone-*.servers.(up|down).
  - openstack.keystone.status - the general status of the Keystone service which is computed using the previous metric and the openstack.keystone.check_api metric.
- Volume (<state> is one of 'available', 'creating', 'attaching', 'in-use', 'deleting', 'backing-up', 'restoring-backup', 'error', 'error_deleting', 'error_restoring', 'error_extending', <service> is one of service is one of 'volume', 'backup', 'scheduler', <service_state> is one of 'up', 'down' or 'disabled')

  - openstack.cinder.volume_creation_time - the time (in seconds) it took to create a new volume.
  - openstack.cinder.volumes.<state> - the number of volumes by state.
  - openstack.cinder.snapshots.<state> - the number of snapshots by state.
  - openstack.cinder.volumes_size.<state> - the total size (in bytes) of volumes by state.
  - openstack.cinder.snapshots_size.<state> - the total size (in bytes) of snapshots by state.
  - openstack.cinder.services.<service>.<service_state> - the total number of Cinder services by state.
  - openstack.cinder.services.<service>.status - status of Cinder services computed from openstack.cinder.services.<service>.<service_state>.
  - openstack.cinder.api.<backend>.status - status of the API services located behind the HAProxy load-balancer, computed from haproxy.backend.cinder-api.servers.(up|down).
  - openstack.cinder.status - the general status of the Cinder service which is computed using the previous metrics and the openstack.cinder.check_api metric.
- Image (<state> is one of 'queued', 'saving', 'active', 'killed', 'deleted', 'pending_delete')
  - openstack.glance.images.public.<state> - the number of public images by state.
  - openstack.glance.images.private.<state> - the number of private images by state.
  - openstack.glance.snapshots.public.<state> - the number of public snapshot images by state.
  - openstack.glance.snapshots.private.<state> - the number of private snapshot images by state.
  - openstack.glance.images_size.public.<state> - the total size (in bytes) of public images by state.
  - openstack.glance.images_size.private.<state> - the total size (in bytes) of private images by state.
  - openstack.glance.snapshots_size.public.<state> - the total size (in bytes) of public snapshots by state.
  - openstack.glance.snapshots_size.private.<state> - the total size (in bytes) of private snapshots by state.
  - openstack.glance.api.<backend>.status - status of the API services located behind the HAProxy load-balancer, computed from haproxy.backend.glance-*.servers.(up|down).
  - openstack.glance.status - the general status of the Glance service which is computed using the previous metric and the openstack.glance.check_api metric.
- Network (<state> is one of 'active', 'build', 'down' or 'error', <owner> is one of 'compute', 'dhcp', 'floatingip', 'floatingip_agent_gateway', 'router_interface', 'router_gateway',

'router_ha_interface', 'router_interface_distributed' or 'router_centralized_snat', <agent_type> is one of 'dhcp', 'l3', 'metadata' or 'openvswitch', <agent_state> is one of 'up', 'down' or 'disabled')
  - openstack.neutron.agents - the total number of Neutron agents.
  - openstack.neutron.networks.<state> - the number of virtual networks by state.
  - openstack.neutron.networks - the total number of virtual networks.
  - openstack.neutron.subnets - the number of virtual subnets.
  - openstack.neutron.ports.<owner>.<state> - the number of virtual ports by owner and state.
  - openstack.neutron.ports - the total number of virtual ports.
  - openstack.neutron.routers.<state> - the number of virtual routers by state.
  - openstack.neutron.routers - the total number of virtual routers.
  - openstack.neutron.floatingips.free - the number of floating IP addresses which aren't associated.
  - openstack.neutron.floatingips.associated] the number of floating IP addresses which are associated.
  - openstack.neutron.floatingips - the total number of floating IP addresses.
  - openstack.neutron.agents.<agent_type>.<agent_state>] the total number of Neutron agents by agent type and state.
  - openstack.neutron.agents.<agent_type>.status - status of Neutron services computed from metric openstack.neutron.agents.<agent_type>.<agent_state>.
  - openstack.neutron.api.neutron.status - status of the API services located behind the HAProxy load-balancer, computed from haproxy.backend.neutron.servers.(up|down).
  - openstack.neutron.status - the general status of the Neutron service which is computed using the previous metrics and the openstack.neutron.check_api metric.
- API response times (<service> is one of 'cinder', 'glance', 'heat', 'keystone', 'neutron' or 'nova', <HTTP method> is the HTTP method name, eg 'GET', 'POST' and so on, <HTTP status> is a 3-digit string representing the HTTP response code, eg '200', '404' and so on)
  - openstack.<service>.http.<HTTP method>.<HTTP status> - the time (in second) it took to serve the HTTP request.

**Ceph**

All metrics are prefixed by ceph.cluster-<name> with <name> is ceph by default.

- Cluster
  - health - the health status of the entire cluster where values 1, 2, 3 represent respectively OK, WARNING and ERROR.
  - monitor - number of ceph-mon processes.
  - quorum - number of quorum members.
- Pools (<name> is the name of the Ceph pool)
  - pool.<name>.bytes_used - amount of data stored in bytes per pool.
  - pool.<name>.max_avail - available size in bytes per pool.

  - pool.<name>.objects - number of objects per pool.
  - pool.<name>.read_bytes_sec - number of bytes read by second per pool.
  - pool.<name>.write_bytes_sec - number of bytes written by second per pool.
  - pool.<name>.op_per_sec - number of operations per second per pool.
  - pool.<name>.size - number of data replications per pool.
  - pool.<name>.pg_num - number of placement groups per pool.
  - pool.total_bytes - total number of bytes for all pools.
  - pool.total_used_bytes - total used size in bytes by all pools.
  - pool.total_avail_bytes - total available size in bytes for all pools.
  - pool.total_number - total number of pools.
- Placement Groups (<state> is a combination separated by + of 2 or more states of this list: creating, active, clean, down, replay, splitting, scrubbing, degraded, inconsistent, peering, repair, recovering, recovery_wait, backfill, backfill-wait, backfill_toofull, incomplete, stale, remapped)
  - pg.total - total number of placement groups.
  - pg.state.<state> - number of placement groups by state.
  - pg.bytes_avail - available size in bytes.
  - pg.bytes_total - cluster total size in bytes.
  - pg.bytes_used - data stored size in bytes.
  - pg.data_bytes - stored data size in bytes before it is replicated, cloned or snapshotted.
- OSD Daemons (<id> is the OSD numeric identifier)
  - osd.up - number of OSD daemons UP.
  - osd.down - number of OSD daemons DOWN.
  - osd.in - number of OSD daemons IN.
  - osd.out - number of OSD daemons OUT.
  - osd.<id>.used - data stored size in bytes.
  - osd.<id>.total - total size in bytes.
  - osd.<id>.apply_latency - apply latency in ms.
  - osd.<id>.commit_latency - commit latency in ms.
- OSD Performance
  - osd-<id>.osd.recovery_ops - number of recovery operations in progress.
  - osd-<id>.osd.op_wip - number of replication operations currently being processed (primary).
  - osd-<id>.osd.op - number of client operations.
  - osd-<id>.osd.op_in_bytes - number of bytes received from clients for write operations.
  - osd-<id>.osd.op_out_bytes - number of bytes sent to clients for read operations.
  - osd-<id>.osd.op_latency - average latency in ms for client operations (including queue time).
  - osd-<id>.osd.op_process_latency - average latency in ms for client operations (excluding queue time).

- osd-<id>.osd.op_r - number of client read operations.
- osd-<id>.osd.op_r_out_bytes - number of bytes sent to clients for read operations.
- osd-<id>.osd.op_r_latency - average latency in ms for read operation (including queue time).
- osd-<id>.osd.op_r_process_latency - average latency in ms for read operation (excluding queue time).
- osd-<id>.osd.op_w - number of client write operations.
- osd-<id>.osd.op_w_in_bytes - number of bytes received from clients for write operations.
- osd-<id>.osd.op_w_rlat - average latency in ms for write operations with readable/applied.
- osd-<id>.osd.op_w_latency - average latency in ms for write operations (including queue time).
- osd-<id>.osd.op_w_process_latency - average latency in ms for write operation (excluding queue time).
- osd-<id>.osd.op_rw - number of client read-modify-write operations.
- osd-<id>.osd.op_rw_in_bytes - number of bytes per second received from clients for read-modify-write operations.
- osd-<id>.osd.op_rw_out_bytes - number of bytes per second sent to clients for read-modify-write operations.
- osd-<id>.osd.op_rw_rlat - average latency in ms for read-modify-write operations with readable/applied.
- osd-<id>.osd.op_rw_latency - average latency in ms for read-modify-write operations (including queue time).
- osd-<id>.osd.op_rw_process_latency - average latency in ms for read-modify-write operations (excluding queue time).

**Pacemaker**

<resource-name> is one of ‘vip__public’, ‘vip__management’, ‘vip__public_vrouter’ or ‘vip__management_vrouter’

- pacemaker.resource.<resource-name>.active - 1 when the resource is located on the host reporting the metric, 0 otherwise.

## R

R is a language and environment for statistical computing and graphics. It is a GNU project which is similar to the S language and environment which was developed at Bell Laboratories (formerly ATnT, now Lucent Technologies) by John Chambers and colleagues. R can be considered as a different implementation of S. There are some important differences, but much code written for S runs unaltered under R. (20, 21, 87)

## Calinski-Harabasz index

A method for identifying clusters of points in a multidimensional Euclidean space is described and its application to taxonomy considered. It reconciles, in a sense, two different approaches to the investigation of the spatial relationships between the points, viz., the agglomerative and the divisive methods. A graph, the shortest dendrite of Florek etal. (1951a), is constructed on a nearest neighbour basis and then divided into clusters by applying the criterion of minimum within cluster sum of squares. This procedure ensures an effective reduction of the number of possible splits. The method may be applied to a dichotomous division, but is perfectly suitable also for a global division into any number of clusters. An informal indicator of the "best number" of clusters is suggested.[**Calinski_Harabasz:1974s**] (21, 87)

## cluster

In statistics it is a group of objects or data values which are grouped on the basis of similarity of some attribute or a value. (21, 87)

## cluster analysis task

Cluster analysis or clustering is the task of grouping a set of objects in such a way that objects in the same group **cluster** are more similar (in some sense or another) to each other than to those in other groups - **clusters**. (20, 87, 90)

## clustering

The same as cluster analysis task (20, 87)

## computer

is a programmable machine that receives input, stores and manipulates data, and provides output in a useful format (87)

## DSL

A domain-specific language (DSL) is a computer language specialized to a particular application domain. This is in contrast to a general-purpose language (GPL), which is broadly applicable across domains, and lacks specialized features for a particular domain. (70, 87)

## Duda-Hart test

presents a statistical test centered around testing the null hypothesis of having c clusters, by comparing with c+1 clusters[](21, 87)

## Eventlet

Eventlet is a networking library written in Python. It achieves high scalability by using non-blocking io while at the same time retaining high programmer usability by using coroutines to make the non-blocking IO operations appear blocking at the source code level. (41, 87)

## Fat Tree

Fat tree DCN architecture handles the oversubscription and cross section bandwidth problem faced by the legacy three-tier DCN architecture.The network elements in fat tree topology also follows hierarchical organization of network switches in

access, aggregate, and core layers. However, the number of network switches is much larger than the three-tier DCN. The fat tree topology offers 1:1 oversubscription ratio and full bisection bandwidth. The fat tree architecture uses a customized addressing scheme and routing algorithm. The scalability is one of the major issues in fat tree DCN architecture and maximum number of pods is equal to the number of ports in each switch. (56, 87)

## FTP

The File Transfer Protocol (FTP) is a standard network protocol used to transfer computer files from one host to another host over a TCP-based network, such as the Internet. (69, 87)

## Greenlet

The “greenlet” package is a spin-off of Stackless, a version of CPython that supports micro-threads called “tasklets”. Tasklets run pseudo-concurrently (typically in a single or a few OS-level threads) and are synchronized with data exchanges on “channels”. (41, 87)

## Hardware support for device passthrough

Both Intel and AMD provide support for device passthrough in their newer processor architectures (in addition to new instructions that assist the hypervisor). Intel calls its option Virtualization Technology for Directed I/O (VT-d), while AMD refers to I/O Memory Management Unit (IOMMU). In each case, the new CPUs provide the means to map PCI physical addresses to guest virtual addresses. When this mapping occurs, the hardware takes care of access (and protection), and the guest operating system can use the device as if it were a non-virtualized system. In addition to mapping guest to physical memory, isolation is provided such that other guests (or the hypervisor) are precluded from accessing it. The Intel and AMD CPUs provide much more virtualization functionality (48, 87)

## HTTP

The Hypertext Transfer Protocol (HTTP) is an application protocol for distributed, collaborative, hypermedia information systems.[11] (69, 87)

## Interface

The term interface port refers to the physical network connector. The term interface or link refers to the logical instance of a network interface port, as seen and configured by the OS. (48, 87)

## iperf

Iperf is a commonly used network testing tool that can create Transmission Control Protocol (TCP) and User Datagram Protocol (UDP) data streams and measure the throughput of a network that is carrying them. Iperf is a tool for network performance measurement written in C. It is a compatible reimplementation of the ttcp program that was developed at the National Center for Supercomputing Applications at the University of Illinois by the Distributed Applications Support Team (DAST) of the National Laboratory for Applied Network Research (NLANR) (71, 87)

## JDBC

JDBC is a Java database connectivity technology (Java Standard Edition platform) from Oracle Corporation. This technology is an API for the Java programming language that defines how a client may access a database. It provides methods for querying and updating data in a database. JDBC is oriented towards relational databases. (69, 87)

## Jitter

In the context of computer networks, jitter is the variation in latency as measured in the variability over time of the packet latency across a network. A network with constant latency has no variation (or jitter). Packet jitter is expressed as an average of the deviation from the network mean latency. However, for this use, the term is imprecise. The standards-based term is *packet delay variation* (PDV). PDV is an important quality of service factor in assessment of network performance. (56, 87)

## k-means algorithm

$k$-means clustering algorithm is a method of vector quantization, originally from signal processing, that is popular for cluster analysis in data mining. k-means clustering aims to partition n observations into k clusters in which each observation belongs to the cluster with the nearest mean, serving as a prototype of the cluster. (21, 87)

## LDAP

The Lightweight Directory Access Protocol (LDAP) is an open, vendor-neutral, industry standard application protocol for accessing and maintaining distributed directory information services over an Internet Protocol (IP) network. (69, 87)

## medoids

Medoids are representative objects of a data set or a cluster with a data set whose average dissimilarity to all the objects in the cluster is minimal[33] (21, 87)

## MongoDB

MongoDB (from humongous) is a cross-platform document-oriented database. Classified as a NoSQL database, MongoDB eschews the traditional table-based relational database structure in favor of JSON-like documents with dynamic schemas (MongoDB calls the format BSON), making the integration of data in certain types of applications easier and faster. (70, 87)

## REST API

In computing, Representational State Transfer (REST) is a software architecture style for building scalable web services.[1][2] REST gives a coordinated set of constraints to the design of components in a distributed hypermedia system that can lead to a higher performing and more maintainable architecture. RESTful systems typically, but not always, communicate over the Hypertext Transfer Protocol with the same HTTP verbs (GET, POST, PUT, DELETE, etc.) which web browsers use to retrieve web pages and to send data to remote servers.[3] REST interfaces usually involve collections of resources with identifiers, for example /people/paul, which can be operated upon using standard verbs, such as DELETE /people/paul. (69, 87)

## $\sigma$

standard deviation (87)

## Silhouette

Silhouette refers to a method of interpretation and validation of consistency within clusters of data. The technique provides a succinct graphical representation of how well each object lies within its cluster. [29] (21, 87)

## SOAP

SOAP, originally an acronym for Simple Object Access Protocol, is a protocol specification for exchanging structured information in the implementation of web services in computer networks. It uses XML Information Set for its message format, and relies on other application layer protocols, most notably Hypertext Transfer Protocol (HTTP) or Simple Mail Transfer Protocol (SMTP), for message negotiation and transmission. (69, 87)

## SR-IOV

This virtualization technology (created through the PCI-Special Interest Group, or PCI-SIG) provides device virtualization in single-root complex instances (in this case, a single server with multiple VMs sharing a device). Another variation, called Multi-Root IOV, supports larger topologies (such as blade servers, where multiple servers can access one or more PCIe devices). In a sense, this permits arbitrarily large networks of devices, including servers, end devices, and switches (complete with device discovery and packet routing). With SR-IOV, a PCIe device can export not just a number of PCI physical functions but also a set of virtual functions that share resources on the I/O device. (48, 87)

**AMQP**

Advanced Message Queuing Protocol (27, 87)

**ARMA**

Auto-Regression Moving Average (20, 87)

**ASA**

Average Speed to Answer (33, 87)

**CRUD**

Create,Read,Update,Delete (87)

**DCN**

Data Center Network (56, 87)

**DHCP**

dynamic host IP addressing (30, 87)

**DUT**

Device Under Test (10, 87)

**ER**

Error rate (33, 34, 38, 87)

**HA**

High Availability (30, 87)

**IaaS**

Infrastructure-as-a-Service (28, 87)

**L3**

layer 3 (30, 87)

**MQ**

message queue (29, 30, 87)

**MTBF**

Mean Time Between Failures (33, 38, 87)

**MTBSI**

Mean Time Between Service Incidents (33, 87)

**MTRS**

Mean Time to Restore Service (33, 38, 87)

**RADOS**

reliable autonomic distributed object store (31, 87)

**REST**

representational state transfer (87)

**RPC**

Remote Procedure Call (27, 87)

**RPO**

Recovery Point Objective (26, 87)

**RPS**

requests per second (33, 36, 87)

**RTO**

Recovery Time Objective (26, 87)

**SLA**

Service Level Agreement (33, 38, 39, 87)

**SOA**

Service Oriented Architecture (27, 59, 87)

**SR-IOV**

Single root input/output virtualization (48, 87)

**TAT**

Turn Around Time (33, 34, 87)

**TSF**

Time Service Factor (33, 87)

**VIF**

Virtual machine network interface (48, 87)

**VM**

Virtual Machine (30, 87)

[1] *ActiveMQ JMeter Performance Testing Tool*. URL: http://activemq.apache.org/jmeter-performance-tests.html (cited on page 66).

[2] Kashif Bilal et al. "A taxonomy and survey on Green Data Center Networks". In: *Future Generation Computer Systems* 36 (2014). Special Section: Intelligent Big Data ProcessingSpecial Section: Behavior Data Security Issues in Network Information PropagationSpecial Section: Energy-efficiency in Large Distributed Computing ArchitecturesSpecial Section: eScience Infrastructure and Applications, pages 189 –208. ISSN: 0167-739X. DOI: http://dx.doi.org/10.1016/j.future.2013.07.006. URL: http://www.sciencedirect.com/science/article/pii/S0167739X13001519 (cited on page 56).

[3] *Cisco UCS Virtual Interface Card 1280*. URL: http://www.cisco.com/c/en/us/products/interfaces-modules/ucs-virtual-interface-card-1280/index.html (cited on page 48).

[4] Douglas E. Comer. *Computer networks and internets (5. ed.)* Pearson Education, 2008. ISBN: 978-0-13-504583-1 (cited on page 56).

[5] "Computing the Performance Measures in Queueing Models via the Method of Order Statistics". In: *Journal of Applied Mathematics* (2011) (cited on page 65).

[6] Richard O. Duda, Peter E. Hart, and David G. Stork. *Pattern Classification (2Nd Edition)*. Wiley-Interscience, 2000. ISBN: 0471056693 (cited on page 21).

[7] OpenStack Foundation. *OpenStack Cloud Administrator Guide*. URL: http://docs.openstack.org/admin-guide-cloud/content/ (cited on page 28).

[8] Karie Gonia. *Latency and QoS for Voice over IP*. URL: https://www.sans.org/reading-room/whitepapers/voip/latency-qos-voice-ip-1349 (cited on page 55).

[9] Brendan Gregg. *Systems Performance: Enterprise and the Cloud*. 1st. Upper Saddle River, NJ, USA: Prentice Hall Press, 2013. ISBN: 0133390098, 9780133390094 (cited on page 48).

[10] Gregor Hohpe and Bobby Woolf. *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2003. ISBN: 0321200683 (cited on page 59).

[11] *Hypertext Transfer Protocol – HTTP/1.1*. URL: https://tools.ietf.org/html/rfc2616 (cited on page 91).

[12] *IP Packet Delay Variation Metric for IP Performance Metrics (IPPM)*. URL: https://tools.ietf.org/html/rfc3393 (cited on page 56).

[13] *ISO 5725-1 Accuracy (trueness and precision) of measurement methods and results*. URL: http://www.iso.org/iso/catalogue_detail.htm?csnumber=11833 (cited on page 10).

[14] *ISO/IEC 20000-1:2011: Information technology – Service management*. URL: http://www.iso.org/iso/catalogue_detail?csnumber=51986 (cited on page 33).

[15] *IT Glossary: Cloud Computing*. URL: http://www.gartner.com/it-glossary/cloud-computing (cited on page 7).

[16] *ITIL v3. Information Technology Infrastructure Library*. URL: https://en.wikibooks.org/wiki/ITIL_v3_(Information_Technology_Infrastructure_Library)/Service_Design#SLA_Structure_Levels (cited on page 33).

[17] Krzysztof Kryszczuk and Paul Hurley. "Estimation of the Number of Clusters Using Multiple Clustering Validity Indices." In: *MCS*. Edited by Neamat El Gayar, Josef Kittler, and Fabio Roli. Volume 5997. Lecture Notes in Computer Science. Springer, Apr. 14, 2010, pages 114–123. ISBN: 978-3-642-12126-5. URL: http://dblp.uni-trier.de/db/conf/mcs/mcs2010.html#KryszczukH10 (cited on page 21).

[18] *Linux virtualization and PCI passthrough*. URL: http://www.ibm.com/developerworks/library/l-pci-passthrough/ (cited on page 48).

[19] Yanchi Liu et al. "Understanding of Internal Clustering Validation Measures". In: *Data Mining (ICDM), 2010 IEEE 10th International Conference on*. 2010, pages 911–916. DOI: 10.1109/ICDM.2010.35 (cited on page 21).

[20] Simon MacMullen. *RabbitMQ Performance Measurements*. URL: https://www.rabbitmq.com/blog/2012/04/17/rabbitmq-performance-measurements-part-1/ (cited on page 65).

[21] John Mandel. *The Statistical Analysis of Experimental Data*. Dover Books on Engineering. Dover Publications, 1984. ISBN: 9780486646664. URL: https://books.google.com/books?id=BQDP5W4xAXUC (cited on page 36).

[22] *MEASURING NETWORK PERFORMANCE: TEST NETWORK THROUGHPUT, DELAY-LATENCY, JITTER, TRANSFER SPEEDS, PACKET LOSS RELIABILITY. PACKET GENERATION USING IPERF / JPERF*. URL: http://www.firewall.cx/networking-topics/general-networking/970-network-performance-testing.html (cited on page 48).

[23] Daniel A. Menasce and Virgilio Almeida. *Capacity Planning for Web Services: Metrics, Models, and Methods*. 1st. Upper Saddle River, NJ, USA: Prentice Hall PTR, 2001. ISBN: 0130659037 (cited on page 36).

[24] Mwtoews. *"Standard deviation diagram" by Mwtoews - Own work, based (in concept) on figure by Jeremy Kemp. Licensed under CC BY 2.5 via Wikimedia Commons*. Feb. 2005. URL: https://commons.wikimedia.org/wiki/File:Standard_deviation_diagram.svg#/media/File:Standard_deviation_diagram.svg (cited on pages 15, 20, 22–24).

[25] *OpenStack Support for Different VIF Types*. URL: https://wiki.openstack.org/w/images/1/1c/Openstack-vif-configuration-SumitNaiksatam-v2.pdf (cited on page 48).

[26] *Oslo messaging simulator*. URL: http://git.openstack.org/cgit/openstack/oslo.messaging/tree/tools/simulator.py (cited on page 66).

[27] *RabbitMQ Performance Tool*. URL: https://github.com/rabbitmq/rabbitmq-perf-html (cited on page 66).

[28] *RFC7231. Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content*. URL: https://tools.ietf.org/html/rfc7231#page-47 (cited on page 34).

[29] Peter J. Rousseeuw. "Silhouettes: A graphical aid to the interpretation and validation of cluster analysis". In: *Journal of Computational and Applied Mathematics* 20 (1987), pages 53 –65. ISSN: 0377-0427. DOI: http://dx.doi.org/10.1016/0377-0427(87)90125-7. URL: http://www.sciencedirect.com/science/article/pii/0377042787901257 (cited on page 93).

[30] Matthew Sackman. *Performance of Queues: when less is more*. URL: http://www.rabbitmq.com/blog/2011/10/27/performance-of-queues-when-less-is-more/ (cited on pages 64, 67).

[31] Matthew Sackman. *Sizing your Rabbits*. URL: http://www.rabbitmq.com/blog/2011/09/24/sizing-your-rabbits/ (cited on page 64).

[32] *SR-IOV-Passthrough-For-Networking*. URL: https://wiki.openstack.org/wiki/SR-IOV-Passthrough-For-Networking (cited on page 48).

[33] Anja Struyf, Mia Hubert, and Peter Rousseeuw. "Clustering in an Object-Oriented Environment". In: *Journal of Statistical Software* 1.4 (Feb. 1997), pages 1–30. ISSN: 1548-7660. URL: http://www.jstatsoft.org/v01/i04 (cited on page 93).

[34] Dr. János Sztrik. *Basic Queueing Theory*. URL: http://irh.inf.unideb.hu/user/jsztrik/education/16/SOR_Main_Angol.pdf (cited on page 64).

[35] *VMware View 4.5 on FlexPod for VMware Design Guide*. URL: http://www.cisco.com/c/en/us/td/docs/solutions/Enterprise/Data_Center/Virtualization/ucs_view_netapp.html (cited on page 48).

## A



## C



## G



## H



## J



## K

## N



## O



## R



## S



## X