MIKELANGELO

D2.4

First Cloud-Bursting Use Case Implementation Strategy

Workpackage:	2	Use Case & Architecture Analysis	
Author(s):	Tzach Livyatan		Cloudius Systems
	Nadav Har'El		Cloudius Systems
Reviewer	Gregor Berginc		XLAB
Reviewer	Holm Rauchfuss		HUA
Dissemination Level	Public		

Date	Author	Comments	Version	Status
2015-08-06	Tzach Livyatan	Initial Draft	V0.0	Draft
2015-08-28	Tzach Livyatan	Ready for review	V0.1	Review
2015-08-31	Tzach Livyatan, Nadav Har'El	Final version	V1.0	Final

Executive Summary

Cloud burst is a sudden increase of incoming traffic to a cloud service. They are common on many useful domains like e-commerce, news, weather information, Internet of Things (IoT), telecommunications and many more. Preparing for a burst is a inherently a hard problem, primarily for two reasons. First, it is typically impossible to predict when the burst will occur and second, costs of maintaining the infrastructure required by such spikes is unjustifiable.

Different approaches have been used to address the bursting problem. This document focuses on handling cloud burst in one specific domain: a NoSQL database Apache Cassandra [15]. Cassandra is a popular, open source fully distributed database, providing high availability with no single point of failure. It excels at fast, low latency writes, with slightly slower reads. Cassandra is often used as part of a cloud service, providing robust support for clusters spanning multiple data-centers.

This report documents the ways of handling cloud bursting in the scope of Cassandra. It details the approach and presents major reasons limiting the Cassandra from supporting more efficient bursting techniques. The report then introduces a fully NoSQL database **Scylla**, fully compatible with the Cassandra client API. Scylla is being implemented on top of the Seastar technology developed partly within MIKELANGELO project. Seastar is an advanced, open-source C++ framework for high-performance server applications on modern hardware. Benchmark results for the Scylla database will be presented demonstrating benefits of using modern APIs taking recent progress in hardware design into consideration.

Table of Contents

Table of Figures

Table of Tables

1 Introduction

1.1 Definition of a cloud burst

A burst is a big jump in throughput in a very short time, which can be 10x, 100x or more of the normal throughput. In the Web domain, such a burst can be the result of an interesting news segment which causes millions to rush to a particular site, or a particular item on sale at Amazon. In the Telco domain a burst can be the result of a sudden event, which causes everyone to make a call at the same time, like an earthquake, or American Idol finale. In the IoT domain, a city wide blackout may cause, once resolved, all the sensors to report at once creating a burst of info coming in. One of the ways bursts are handled is prioritisation of traffic. For example, during an emergency situation, calls directly related to the emergency (112 calls and communication between emergency management team) have higher priority than all other calls. However, the net neutrality requires no such prioritisation when it comes to the Internet. Consequently, handling the burst is in full control of the service provider.

In many cases, bursts are completely unexpected, making it very hard to prepare for them in advance. There are two fundamental approaches to handle a burst, or any fast increase in required capacity:

1. Increase the capacity as fast as possible
2. Limit incoming traffic, either by throttling the clients, or dropping messages

Clearly, the first is better as it does not affect users. However, it is typically impossible to handle expansion of the infrastructure instantaneously resulting in service unavailability resulting in connection timeouts. Therefore, in many cases both these methods need to be applied, sometime at the same time. This document primarily focuses on the first, although, the second might kick in as well, if the incoming traffic rate grows faster than the cluster capacity.

Naturally, preparing for a burst is a hard problem, as one does not want, or can not, have all the required capacity to support extra traffic in advance. This would require maintenance of spare capacity that would not provide a better service level in normal operation mode. Therefore, costs of this spare infrastructure are unjustifiable in most common use case of a general-purpose service provisioning where burst events are rare and unpredictable.

This document focuses on handling cloud burst of a modern NoSQL database implementation. Apache Cassandra [15] is used as a baseline database. Cassandra is an open source distributed database designed to handle large amounts of data across many commodity servers, providing high availability with no single point of failure. It excels at fast, low latency writes, with slightly slower reads. Cassandra is often used as part of a cloud service, providing robust support for clusters spanning multiple data-centers.

Cassandra is used in all of the aforementioned domains, making it ideal for addressing and consequently solving some of the crucial pain points present in current implementation.

Although this use case focuses only on Cassandra database it is worth mentioning that other implementations suffer from the same or very similar deficiencies.

Examples for cloud service which use Cassandra are:

- Apache UserGrid[12] uses Cassandra as a database for BaaS (backend as a service)
- Instaclustr [13] provides Cassandra as a service
- Netflix [14] uses Cassandra as a backend for many of its services

There are two ways Cassandra, or any other database, can handle a burst by increasing the capacity and not refusing requests:

1. **Scale out the cluster by adding more hardware.**

   In most cases, "hardware" is virtual, which can be allocated relatively quickly on private or public cloud providers.

2. **Multi-tenancy - using one big clusters for many different users.**

   This method is requires allocating extra nodes in advance. These nodes are active members of the cluster handling the normal workload of Cassandra databases causing the entire database cluster to be under committed. However, this redundant resources will immediately help resolving a sudden increase in traffic of one database. Unlike the problem description above, costs of these extra nodes are spread between many users and are, consequently, manageable.

   Multi-tenancy requires strong isolation between users, making sure no rogue user or operation can affect other users' load. Each of the resources (CPU, IO, memory, and bandwidth) must be tightly controlled and limited per user.

1.2 Cassandra

Apache Cassandra [15] is an open source distributed database designed to handle large amounts of data across many servers, providing high availability with no single point of failure. Commodity servers may be used in place of high-end compute/storage nodes.

Cassandra is often used as a cloud service, providing robust support for clusters spanning multiple data-centers.

Among all SQL and NoSQL databases available today, Cassandra is considered the best choice for highest throughput and scalability. Study shows [2] it outperforms Redis, HBase, VoltDB, Voldemort and MySQL. Cassandra cluster can grow to thousands of nodes with no single point of failure.

NoSQL databases, like Cassandra, are often categorized along two axes:

- Data model:
  - Key Pair DB (like Redis, AWS DynamoDB),
  - Document DB (like Mongo),

  - Wide Row, or columnar, row store, DB (HBase and Cassandra), and
  - Graph DB (like Titandb).
- Consistency, Availability and Partition tolerance:
  - Consistent and Available (CA),
  - Available and Partition-tolerant (AP) or
  - Consistent and Partition-tolerant (CP).

Cassandra uses Wide Column data model and is Available and Partition-tolerant (like DynamoDB, Riak). According to the CAP theorem [18], having all three properties is impossible. These two properties of the Cassandra database are presented in the next subsections.

1.2.1 Wide Column data model

A wide column store is a type of key-value database. It uses tables, rows, and columns, but unlike a relational database, the names and format of the columns can vary from row to row in the same table. Unlike strict key-value, Wide Column supports a second level of indexing inside each row.

Key-Value can be described as a map: key -> value

Wide Column can be described as a map of maps: partition key -> clustering key -> value

Cassandra's data model is a partitioned row store: Rows are organized into tables where the first component of a table's primary key is the partition key. A partition key is a mapping between fields in the table (called primary index) and nodes which store them. Within a partition, rows are clustered by the remaining columns of the primary key.

Table 1 introduces a simplified schema to demonstrate partition and clustering keys [3].

Table 1: Creating a table in Cassandra, with weatherstation_id as partition key and event_time is clustering key.

```
CREATE TABLE temperature (
weatherstation_id text,
event_time timestamp,
temperature text,
PRIMARY KEY (weatherstation_id,event_time)
);
```

This schema will support inserts, as presented in Table 2.

Table 2: Inserting data into a Cassandra table.

```
INSERT INTO temperature(weatherstation_id,event_time,temperature)
VALUES ('1234ABCD','2013-04-03 07:01:00′,'72F');
```

The schema will further support queries, as presented in Table 3.

Table 3: Query data from a Cassandra table.

```
SELECT temperature
FROM temperature
WHERE weatherstation_id='1234ABCD'
AND event_time > '2013-04-03 07:01:00′
AND event_time < '2013-04-03 07:04:00′;
```

Note that range queries are possible only on clustering keys and not on the partition key. This is because partition keys are a result of a hash function on fields of the table. The result of the hash, determine on which node the row will be stored, making sure rows are evenly spread across nodes.

Clustering keys use the actual value for ordering and located on the same row and node.

1.2.2 Availability

Cassandra uses eventual consistency, a consistency that informally guarantees that, if no new updates are made to a given data item, eventually all accesses to that item will return the last updated value.

Eventual consistency refers to a strategy used by many distributed systems to improve query and update latencies, and in a more limited way, to provide higher availability to a system than could otherwise be attained.

Cassandra's eventual consistency, as opposed to strict consistency of other DB (SQL, HBase), is what is making it AP (Availability and Partition tolerance). Eventual Consistency reduces the Consistency requirement by allowing data to be synchronized at a later time. This makes the system more available, as not all nodes need to have the same copy of the database all the time.

Cassandra further extends the concept of eventual consistency by offering a so called tunable consistency. For any given read or write operation, the client application decides how consistent the requested data must be. Consistency levels in Cassandra can be configured to manage availability versus data accuracy [16] [17]

2 Cassandra Cloud Burst

In the previous chapter we have explained what the cloud burst is, what the Cassandra DB is, and why handling the burst in Cassandra is so important. This chapter will first describe how Cassandra handles the cloud burst by presenting empirical results of the initial benchmark. We will then present Scylla, a new NoSQL database and its approach to solving Cassandra's issues with the cloud burst.

2.1 Current Limitations

As previously discussed, there are two main ways to handle burst in general, namely fast scaling and multi-tenancy. Cassandra addresses both of these options, however with significant limitations presented next.

2.2 Fast Scaling

Cassandra can only support one server joining the cluster at a time. Trying to start and join more than one node will end up with the following error message (Table 4).

Table 4: Cassandra error when adding more than one node to a cluster at the same time

ERROR [main] 2015-04-29 12:42:41,620 CassandraDaemon.java:465 - Exception encountered during startup java.lang.UnsupportedOperationException: Other bootstrapping/leaving/moving nodes detected, cannot bootstrap while cassandra.consistent.rangemovement is true INFO [StorageServiceShutdownHook] 2015-04-29 12:42:41,622 Gossiper.java:1318 - Announcing shutdown

Even Cassandra best practices recommend that the minimum time between joining two nodes to the cluster is 2 minutes [4].

2.3 Multi-tenancy

Cassandra does not fully support multi-tenancy. A "key-space" can be used to group several tables under one scope and there is some support for access control (with the GRANT command). However support for resource accounting (for example for billing the individual tenants) and resource isolation is very limited. A client performing too many read or write operations to one key-space, will cause performance degradation in clients of other key-spaces on the same cluster.

2.4 Testbed

To evaluate Cassandra behaviour under cloud burst, we executed a benchmark designed to measure Cassandra scale up speed. This section explains the set-up of the benchmark, followed by the initial benchmark results.

2.4.1 Test Setup

Test setup consists of the following node types

- Cassandra nodes: m3.large (each node has 2 vcpus, and 7.5 GB of RAM)
- Loaders nodes: c3.8xlarge

All nodes are running Linux, Ubuntu 14.04 and Cassandra version 2.1.8.

Our plan is to use the same setup later in the project to compare new Cassandra updates as well as Scylla database.

Our own grown Ansible framework [6] takes care of launching the instances, setting up the Cassandra cluster, running the loaders and collecting the results. The framework will be open sourced in the next two months.

2.5 Test Execution

Each of the loaders (2 in this case) runs a process cassandra-stress [5]'s "write" workload.

We start from a Cassandra cluster consisting of 2 nodes, adding a new node to the cluster at least 2 minutes apart (more on this above), until a total of 10 nodes is reached.

The commands in Table 5 were used to run the test.

Table 5: Using the test frame work to run scale up benchmark.

./ec2-setup-cassandra.sh -e "cluster_nodes=2" -e "instance_type=m3.large"
./ec2-setup-loadgen.sh -e "load_nodes=2" -e "instance_type=c3.8xlarge" -e "install=false"
./ec2-stress.sh 1 -e "load_name=init.v1" -e "stress_options='-errors ignore'" -e "command_options=duration=3m" -e "threads=750" -e "clean_data=true"
~19-20K per loader
./ec2-add-node-to-cluster.sh -e "cluster_nodes=8" -e "stopped=true" -e "instance_type=m3.large" -e "install=false"
./ec2-stress.sh 1 -e "load_name=scale_from_2_to_10.m3.large.v1" -e "stress_options='-errors ignore'" -e "command_options=duration=45m" -e "threads=750" -e "clean_data=true" -v 2>&1 | tee stress.log & sleep 600 ; ./ec2-start-server.sh 8

2.6 Benchmark Results

Figure 1 presents the throughput, as sum of what was measured by the loaders during the test.

The drops of throughput when adding new nodes are clearly visible in the graph. After a short period of time during which the stabilization of the cluster occurs, this drop is restored and the obvious gain is visible for each additional node.

Axes of the following charts are as follows

- X axis is the time, with 10 seconds between each point.
- Y axis is either the number of operations (requests) per second for chart 1 or latency in milliseconds for charts 2,3

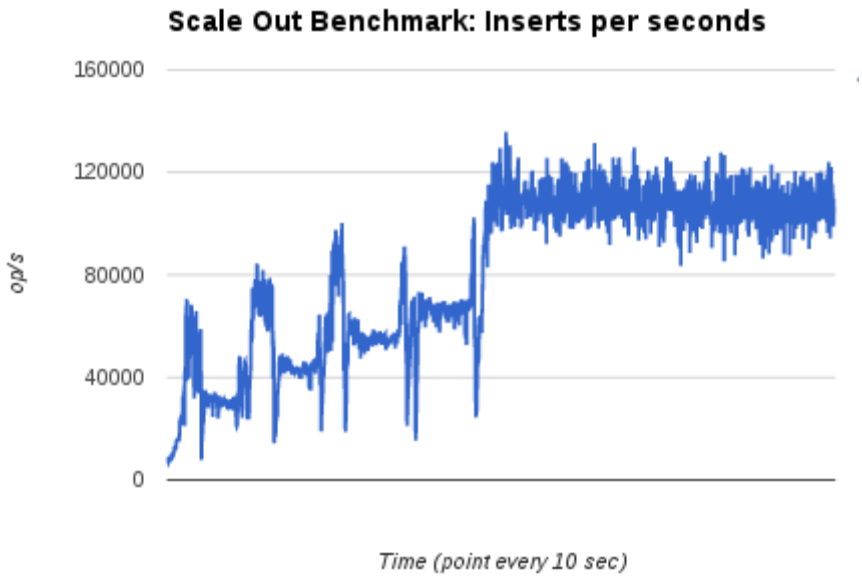

Figure 1: Scale out benchmark: Inserts per second

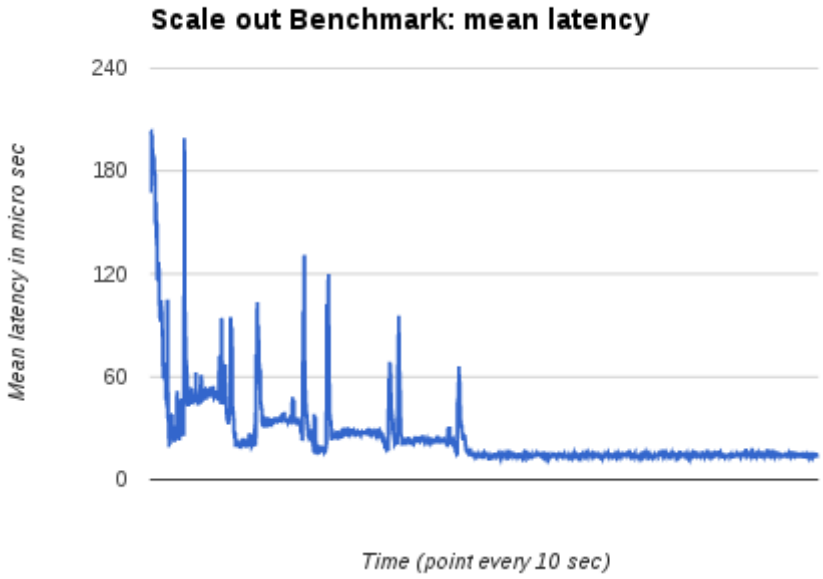

Figure 2: Scale out benchmark: mean latency

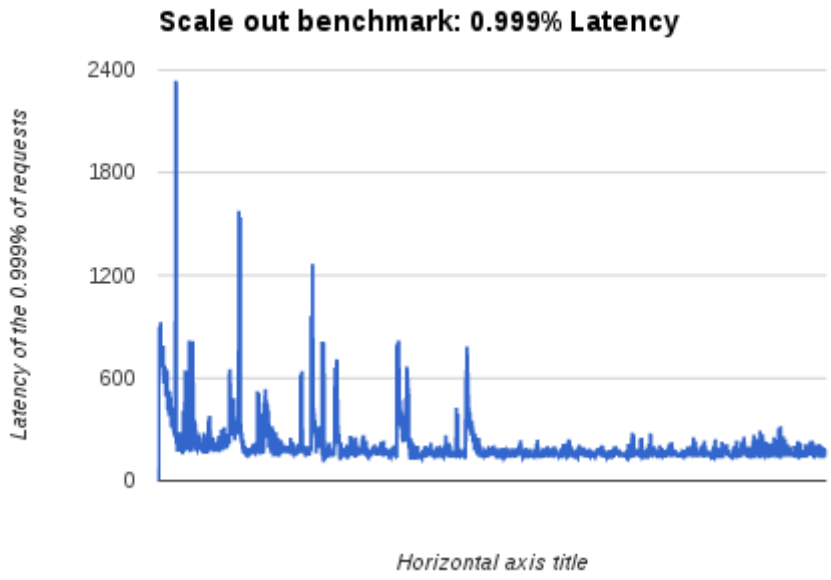

Figure 3: Scale out benchmark: 0.999% Latency

Figure 2 presents mean latency of Cassandra insert request for the same test in milliseconds, and Figure 3 shows latency of the worst 0.001% of the requests.

Full results and raw data are available at [7].

3 Scylla and Seastar

With the introduction of Cassandra and its drawback we are now focusing on solutions. First we present a novel framework for developing high performance applications called Seastar. We then introduce **Scylla**, a new Cassandra compatible NoSQL DB built on top of **Seastar** demonstrating its capabilities for general application.

3.1 Seastar

Seastar [10] is an advanced, open-source C++ framework for high-performance server applications on modern hardware.

Seastar is the first open-source framework to bring together a set of extreme architectural innovations considering recent advances in modern hardware. These includes:

- Shared-nothing design: Seastar uses a shared-nothing model that shards all requests onto individual cores.
- High-performance networking: Seastar offers a choice of network stack, including conventional Linux networking for ease of development, DPDK for fast user-space networking on Linux, and OSv network stack on OSv.
- Message passing: a design for sharing information between CPU cores without time-consuming locking.
- Futures and promises: an advanced new model for concurrent applications that offers C++ programmers both high performance and the ability to create comprehensible, testable high-quality code. Futures and promises are providing the high-level API to third party developers to easily take advantage of the Seastar framework.

While many applications can benefit from high performance, Seastar is currently focused on high-throughput, low-latency I/O intensive applications. In the scope of the MIKELANGELO project, Seastar will power a novel NoSQL database Scylla, Apache Cassandra client API compatible. However, the Seastar framework can be useful for other applications in the MIKELANGELO scope. As examples, two sample Seastar application are available:

- Seastar HTTPD: simple web server
- Seastar Memcached: a fast server for the Memcached key-value store [11]

Seastar framework will be presented in more details in deliverable D2.16 The first OSv guest operating system MIKELANGELO architecture.

3.2 Scylla

Scylla is a new column-store database, fully compatible with Apache Cassandra [15]. It is based on the Seastar framework briefly presented in the previous section [10], and designed to embrace shared-nothing approach with a core-granularity, running multiple engines, one per core, each with its own dedicated memory, CPU and multi-queue NIC.

Scylla reuses Cassandra scalability features used between servers, and uses them between cores in the one node. This, results in each node being an independent (as much as possible) element, without changing the node's external interfaces, and without exposing the internal number of cores.

Scylla will improve the two main shortcomings of Cassandra for the two burst handling methods described above:

- **Fast Scaling**

  Scylla's superior throughput is going to reduce resource consumption while handling data streaming as well as serving all incoming traffic. Scylla implements a more efficient data streaming protocol, which will reduce the time required to redistribute the data to the new, bigger cluster. Scylla will significantly reduce Cassandra limitation of 2 minutes wait between adding nodes.

- **Multi-tenancy**

  Scylla will support real multi-tenancy, including explicit limitation on each of the following resources: CPU usage, disk usage and networking bandwidth and memory.

  This will allow multiple users to share a resource pool larger than required by each individual tenant. This is going to practically remove the time required to add hardware during the burst because the extra capacity will always be there, ready for any one of the users to burst. This will increase the efficiency and agility of a DB deployment, in the same way cloud added agility to independent application deployment. Consequently, all these tenants will also share the costs of the extra capacity.

3.3 Expectations for the MIKELANGELO Stack

We expect the MIKELANGELO project will provide:

- **Seastar library**

  Profiling Cassandra has proven that traditional socket APIs were decimating its performance. During preliminary investigation we have discovered that approximately 20% of the run time is consumed by lock attempts (most of these uncontested locks, which still take time).

  Moreover, we saw significant CPU idle time because of the coarseness of Cassandra's threads. Those threads were needed because of the blocking nature of memory-mapped files, which are in turn necessary because of limitations of Java (in which Cassandra is written).

  Seastar aims to solve this problem by building a new API for writing modern super-efficient asynchronous applications on many-core VMs.

Scylla will serve as a proof-of-concept application, using Seastar to implement Cassandra compatible NoSQL DB.

- **OSv OS**

  OSv will reduce the boot time and thus the time required adding new nodes to the cluster (scale-out). Shorter boot time is both due to faster kernel boot and smaller virtual images.

- **Hypervisor sKVM**.

  In our initial experiments with Scylla, we used PCI device assignment (a.k.a. SR-IOV) for the network card virtualization. The reason for using device assignment, and not the more popular and more flexible paravirtual network-card mechanism, was device assignment's superior performance - higher throughput and lower latency. However, with Mikelangelo's sKVM's exit-less paravirtual network card, we anticipate that we would be able to have the best of both worlds: use a paravirtual network card (namely, virtio in KVM), and do so without giving up on the optimal performance of the guest. Seastar (and therefore also Scylla) already supports virtio network cards (like the one supplied by sKVM), to allow its integration with sKVM.

  Moreover, sKVM's IOcm will allow multiple such high-throughput Seastar-based guests to reside on the same host while each maintains its peak performance, and while wasting minimal additional host resources: the IOcm will automatically determine the number of cores the host will need to dedicate to the host side (backend) of network virtualization.

4 Use Case Set-up

As described above, the planned setup is a cluster of 10 Cassandra/Scylla machines, with at least 2 loaders. We may need to increase the number of loaders as 2 loaders might not be enough to overload Scylla servers (note the loaders are running on much more powerful machines than the cluster nodes). See the Testbed section above for more details.

4.1 Physical Hardware

Test above was done with a cluster of 10 nodes, each with 2 vCPUs, and additionally 2 loaders each with 32 vCPUs. We plan to use similar setup to test Scylla.

4.2 Software

Unlike Cassandra which is Java based and use the JVM, Scylla is native and does not use a run-time VM. Scylla leverages a modified DPDK stack from Intel [8].

The stress tool, cassandra-stress is also implemented in Java.

As mentioned above we will use the Ansible based framework to set up and run the benchmarks, of both Cassandra and Scylla.

4.3 Data

We will test Cassandra/Scylla using standard Cassandra testing tool, cassandra-stress, for reads, writes, and a mix of reads and writes.

If required, the tool will allow us to create specific testing profiles, which are more tailored to the needs of the project, and the partners.

4.4 Mandatory Requirements

The following requirements are mandatory for our use case:

- Implementation of Scylla, a Cassandra compatible, Seastar base DB.
- Implementation of testing framework
- Benchmark and compare
    - Cassandra on Linux
    - Cassandra on OSv
    - Scylla/Seastar

In addition the following MIKELANGELO features [19] are required:

- 8 Hypervisor support for CentOS guest
- 10 Hypervisor support for Ubuntu guest
- 9 Hypervisor support for OSv guest
- 55 Hypervisor virtual network conectivity vhost-net should be able to connect to a virtual switch

- 56 Hypervisor physical network connectivity
- 58 Hypervisor should be able to run Ubuntu guest with a virtio-net virtual NIC
- 42 Capture performance metrics of guest OS - OSv
- 43 Services/Applications Monitoring
- 20 OSv support environment variables
- 23 OSv support / API for monitoring the instances
- 29 OSv support multi-threading

4.5 KPIs

For the cloud bursting use case, the following KPIs will be measured:

- Time until a cluster, continuously being under stress, grow from 2 nodes to 10, and stabilize (the maximum number of nodes may be changed in later stages of the project). In this context the cluster is stable when data was redistributed, and all data streams are done.
- Time until the cluster can handle 1,000,000 requests per second.

  We may lower this number, not to exceed the 10 cluster size for Cassandra.

  If we choose a lower value, we will use it for both Cassandra and Scylla.

- Number of nodes needed to handle this load.

These KPIs are important for the first version of this deliverable. They will help in evaluation of the relative improvement of throughput per node, and the actual ability to handle cloud burst.

In later phases of the project we are going to extend the list of KPIs and metrics supporting KPIs to thoroughly evaluate the entire MIKELANGELO. For example, a number of changes required within OSv kernel for supporting this use case will be reported and various relative improvements will be measured and reported.

5 Implementation Plan

We plan a three-step improvement:

1. Run Cassandra on OSv

   OSv kernel is very small making virtual images containing user applications significantly smaller than other guest OSes. Smaller footprint will speed up the acquisition of virtual images and improve boot time significantly.

2. Replace Cassandra with Scylla, a Seastar-based clone.

   Scylla will significantly improve throughput and latency of Cassandra, allowing the same number of nodes to support much higher traffic. This will further reduce the need to large scale outs. Even in case scale out will be required Scylla will improve the overall time required thanks to much shorter boot times and stabilisation phases.

3. Adding multi-tenancy features to Scylla

   Users sharing the same underlying database cluster, allowing a bigger cluster, which in turn can endure the increase of traffic during the burst.

5.1 High Level Time Plan

The following is a high level plan for implementing various components of this use case.

- Basic Scylla functionality - End of the year (2015)
- Test Scylla - 2016 Q1
- Compare Cassandra to Scylla for the cloud burst use case - 2016 1Q
- Add Multi Tenancy to Scylla - sometime in 2016
- Compare Cassandra to Scylla with Multi Tenancy for the cloud burst use case - 2016

6 Evaluation and Validation Plan

We will evaluate three deployments:

1. Out of the box, Cassandra Apache on Linux
2. Cassandra Apache on OSv
3. Scylla

For each of the use cases we will evaluate cloud burst use case, using the following KPIs:

1. Time until a cluster, continuously being under stress, grow from 2 nodes to 10, and stabilize.
2. Time until the cluster can handle 1,000,000 requests per second.

Our expectation is to improve on both KPIs, from the first deployment (Apache Cassandra on Linux) to the second and third. As describe above, we already measured the first KPI for the first deployment.

7 Concluding Remarks

As described, cloud burst is a real world use case, affecting critical cloud based systems. In this document we described how one particular component, the database, of a cloud based system handles burst, focusing on Cassandra, one of the most popular NoSQL DB.

We described two ways to handle cloud burst, scaling up and multi tenancy, and why Cassandra lacks in implementing both.

We than introduced a new Cassandra client API compatible database, Scylla, based on the high performance framework Seastar, and explained why Scylla and Seastar will improve Cassandra for the cloud burst use case. We have also proposed KPIs supporting measurable improvements and introduced a testing environment and framework to execute these tests.

Our preliminary tests already show there is much room for improvement in Cassandra cloud burst use case.

8 References and Applicable Documents

[1] The MIKELANGELO project, http://www.mikelangelo-project.eu/
[2] http://vldb.org/pvldb/vol5/p1724_tilmannrabl_vldb2012.pdf
[3] https://academy.datastax.com/demos/getting-started-time-series-data-modeling
[4] http://docs.datastax.com/en/cassandra/2.1/cassandra/operations/ops_add_node_to_cluster_t.html
[5] http://docs.datastax.com/en/cassandra/2.1/cassandra/tools/toolsCStress_t.html
[6] https://github.com/cloudius-systems/ansible-cassandra-cluster-stress
[7] https://docs.google.com/spreadsheets/d/17gn5R9w31fctzx-mwZXeFrdmk6kBvVhNRHD-P5kcqow/edit#gid=1223617871
[8] http://dpdk.org/
[9] https://cassandra.apache.org/doc/cql3/CQL.html
[10] http://www.seastar-project.org/
[11] http://www.seastar-project.org/memcached/
[12] http://usergrid.apache.org/
[13] https://www.instaclustr.com/
[14] http://techblog.netflix.com/search/label/Cassandra
[15] https://cassandra.apache.org
[16] http://www.datastax.com/dev/blog/your-ideal-performance-consistency-tradeoff
[17] http://docs.datastax.com/en/cassandra/2.0/cassandra/dml/dml_config_consistency_c.html
[18] Armando Fox and Eric Brewer, “Harvest, Yield and Scalable Tolerant Systems”, Proc. 7th Workshop Hot Topics in Operating Systems (HotOS 99), IEEE CS, 1999, pg. 174-178.
[19] D2.19 The first MIKELANGELO architecture