BIOBANKCLOUD
Your PaaS for Biobanking

Project number:	317871
Project acronym:	BIOBANKCLOUD

WORK PACKAGE 2 :

SCALABLE STORAGE

Work Package Leader Name and Organisation:

Jim Dowling, KTH – Royal College of Technology (KTH)

E-mail: jdowling@kth.se

PROJECT DELIVERABLE

D2.1: Highly Available HDFS

Deliverable Due date (and month since project start): 2013-11-30, m12

Document history

Version	Date	Changes	By	Reviewed
0.1	2013-11-23	First version	Salman Niazi Kamal Hakimzadeh Alberto Lorente Mahmoud Ismail	Jim Dowling

Executive Summary

This deliverable consists of a software deliverable of the highly available Hadoop Filesystem (HDFS), a userguide for the software, and a short description of the system's architecture.

Our implementation of HDFS provides a new distributed model for HDFS' metadata, based on storing the metadata in MySQL Cluster, a distributed, in-memory, highly available relational database. Our implementation strengthens the replication model of HDFS v2, which is based on eventually consistent primary-secondary replication, to one of shared atomic memory, thus simplifying some of HDFS' internal protocols and enabling support for many NameNodes (as opposed to only a primary and secondary NameNode in HDFS v2). Our implementation also maintains the consistency semantics of HDFS, and we validate this by ensuring that all 300+ unit tests for HDFS pass.

This deliverable also describes the platform-as-a-service (PaaS) support we provide for our HDFS implementation. Our HDFS implementation, along with Apache YARN, can be easily installed by unsophisticated users by just pointing and clicking from our portal website to any of the following platforms: Amazon Web Services, OpenStack or a cluster of (bare-metal) hosts. We also provide a Dashboard to administer and monitor the deployed Hadoop cluster.

The document is structured as a userguide for installing and managing a Hadoop platform containing our highly available HDFS distribution, followed by a brief description of the system architecture.

The code is available for download now, although it is still very much beta and under heavy development.

Table of Contents

List of Figures

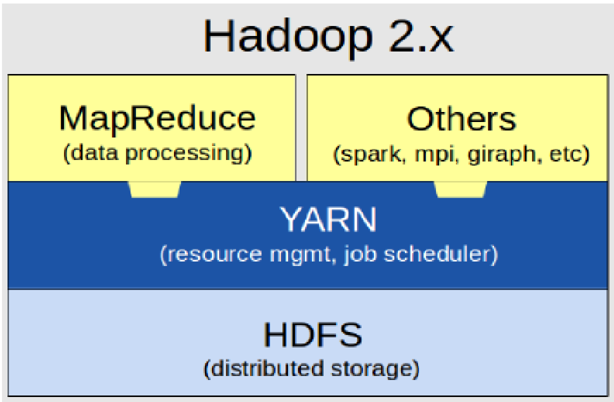
List of Examples

Chapter 1. Hadoop Open PaaS

Highly Available Hadoop Filesystem (HDFS)

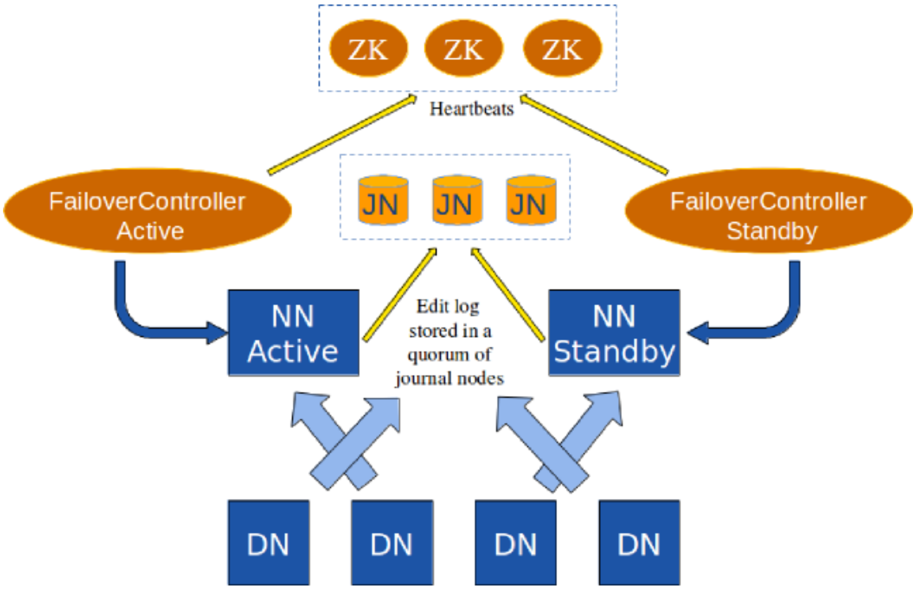
Due to the rapid growth of data in recent years, distributed file systems have gained widespread adoption. The new breed of distributed file systems reliably store petabytes of data, and also provide rich abstractions for massively parallel data analytics. The Hadoop Distributed File System (HDFS) [3] is a distributed, fault-tolerant file system designed to run on low-cost commodity hardware that scales to store petabytes of data, and is the file storage component of the Hadoop platform. HDFS provides the storage layer for MapReduce, Hive, HBase, Spark and all other YARN applications, see Figure 1.1, "Hadoop v2".

Figure 1.1. Hadoop v2



In 2013, HDFS v2 [14] introduced a new highly available metadata architecture, where, as in HDFS v1, the entire filesystem's metadata is stored in memory on a single node, but in v2 changes to the metadata (*edit log entries*) are now replicated and persisted to a set of (at least three) Journal Nodes using a quorum-based replication algorithm. In HDFS v2, a Primary and Secondary NameNode can be configured, where the Primary NameNode is responsible for managing the metadata, and the Secondary NameNode keeps an eventually consistent copy of the metadata. The Secondary NameNode is kept in sync with the Primary by two mechanisms: firstly, by asynchronously applying all edit log entries that have been committed at the Journal Nodes, and secondly, receiving the same set of heartbeats from Data Nodes that are received by the Primary. The Primary/Secondary replication model is also known as an Active/Standy or Master/Slave replication model, and was popularized by databases in the 1990s. HDFS' implementation of this eventually consistent replication model is more limited than in the traditional relational database world, as all read and write requests are sent to the Primary. In typical Master/Slave configurations, writes are sent to the master, while reads are load-balanced across slaves. The reason all write requests are sent to the Primary to ensure a single consistent copy of the metadata. Read requests are also sent to the Primary, as reads at the Secondary could result in operations being executed on stale metadata. This is a bigger problem for a filesystem, such as HDFS, than it would be for a Web 2.0 social media application with non-critical data, using a Master/Slave database setup. Thus, reads are only sent to the Primary. If the Primary fails, however, the Secondary needs to take over as Primary. Before it can take over, it first has to catch up with the set of edit log entries applied to Primary before it failed. The period of time before all outstanding edit log entries are applied at the Secondary before it can take over may be up to tens of seconds, depending on the current load of the system and the hardware and software setup. Another limitation of the Primary/Secondary model, is that client and Data Nodes from HDFS need to have a consistent view of who the current Primary NameNode is. They do this by asking a Zookeeper coordination service that needs to run on at least 3 nodes to provide a fault tolerant reliable service. Finally, the concurrency model supported by HDFS v2 is still multiple-readers, single-writer.

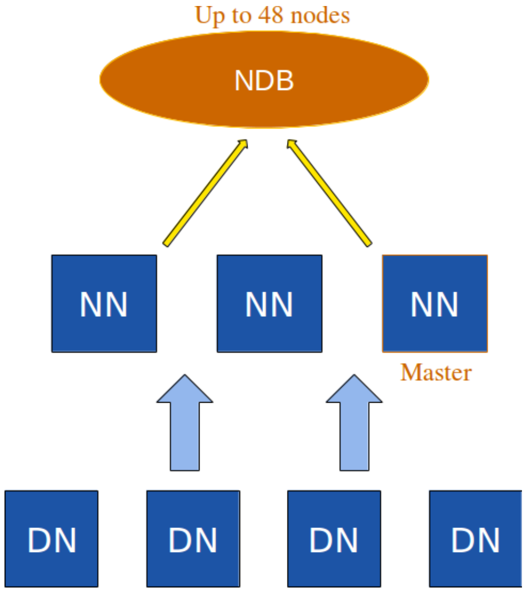
Figure 1.2. HDFS v2 NameNode Primary/Secondary Replication Model



Hop HDFS

In contrast, our implementation of HDFS, called *Hop HDFS*, replaces the Primary-Secondary metadata model with shared, transactional memory, implemented using a distributed, in-memory, shared-nothing database, MySQL Cluster (NDB) [11] see figure Figure 1.3, “Hop HDFS”. MySQL Cluster is a real-time, ACID-complaint transactional database, with no single point of failure, that has predictable, milisecond response times with the ability to service millions of operations per second. By leveraging MySQL Cluster to store HDFS' metadata, the size of HDFS' metadata is no longer limited to the amount of memory that can be managed on the heap of JVM on a single node. Our solution involves storing the metadata in a replicated, distributed, in-memory database that can scale up to several tens of nodes, all while maintaining the consistency semantics of HDFS. We maintain the consistency of the metadata, while providing high performance, all within a multiple-writer, multiple-reader concurrency model. Multiple concurrent writers are now supported for the filesystem as a whole, but single-writer concurrency is enforced at the inode level. Our solution guarantees freedom from deadlock and progress by logically organizing inodes (and their constituent blocks and replicas) into a hierarchy and having transactions defining on a global order for transactions acquiring both explicit locks and implicit locks on subtrees in the hierarchy; this solution is motivated by [9] and [2]. The use of a database, however, also has its drawbacks. As the data now resides on remote hosts on the network, an excessive number of roundtrips to the database harms system scalability and increases per-operation latencies. We ameliorate these problems by introducing a snapshotting mechanism for transactions, where, at the beginning of a transaction, all the resources it needs are aquired in the defined global order, while simulatenously taking row-level locks for those resources. On transaction commit or abort, the resources are freed. This solution enables NameNodes to perform operations on a local copy (or snapshot) of the database state until such time as the transaction is completed, thus reducing the number the number of roundtrips required to the database, see Figure 1.10, “Reduction in the number of DB roundtrips by snapshotting” .

Figure 1.3. Hop HDFS



Leader Election

In HDFS, there are a number of background tasks that are problematic if multiple NameNodes attempt to perform them concurrently. Examples of such tasks include:

1. replication monitoring,
2. lease management,
3. block token generation,
4. and the decomissioning of datanodes.

Without any coordination between NameNodes, and since all NameNodes have identical behaviour, we would have problems with many of these background tasks. For example, if a block becomes under-replicated, several NameNodes could independently identify this under-replicated state, and each would select a DataNode to replicate that block to. This would cause multiple re-replications of the block, leading it to enter an over-replicated state, upon which, multiple NameNodes could recognize this over-replicated state and attempt to remove replicas, possibly leading to an under-replicated state - back where we started. We solve this coordination problem, by implementing a Leader Election Algorithm, where only the leader NameNode is assigned the task of performing the above background tasks. Our leader election algorithm uses the shared, transactional memory abstraction provided by MySQL Cluster to coordinate the election process.

Definition: Correct NameNode A correct NameNode is defined as an alive process that is is able to write to NDB in a bounded time interval.

A leader NameNode is a NameNode from the set of NameNodes that is correct and is responsible for listening to DateNode heartbeats and assigning various tasks to them as well as responsible for managing background tasks. In a deployed system, the bounded time interval during which it is expected that a NameNode can write to NDB is typically set to around 1 second. The properties that our leader election algorithm guarantee are:

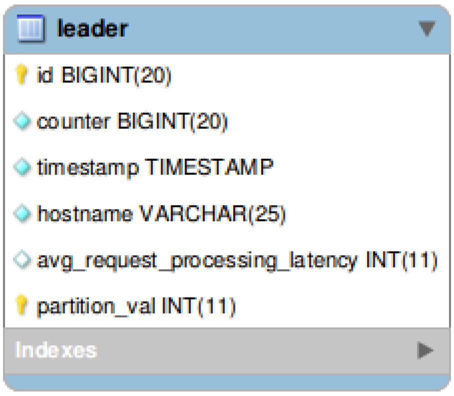
1. *Completeness:* after a bounded time interval, all correct NameNodes will detect every NameNode that has crashed.
2. *Agreement:* after a bounded time interval, all correct NameNodes will recognize one among them as the leader. All will agree to the same NameNode being the leader.

3. *Stability:* If one correct NameNode is the leader, all previous leaders have crashed

Leader Election Algorithm

The leader election algorithm runs continuously at the NameNodes in rounds. Each NameNode is assigned an (integer) id. At any point in time, the correct NameNode with the lowest id is elected as the leader. The central focus of the algorithm is to detect failures and elect a new leader using heartbeats. We implement heartbeats as counters in a table in NDB. The algorithm is run at NameNodes in rounds (every nth second), and it is expected that each NameNode will send only one heartbeat per round, although our algorithm tolerates minor deviations, as explained later. NameNodes can discover who the leader is by reading the ids of correct NameNodes from a table in NDB. NDB's shared transactional memory allows all NameNodes to have a uniform view of the correct NameNodes in the system. We implemented our heartbeat model using the schema shown in Figure 1.4, “Leader Table in NDB”. Some sample records are also shown to show an example run of the algorithm.

Figure 1.4. Leader Table in NDB



Once a NameNode starts, it starts a rounds timer which periodically triggers causing it to send a heartbeat to the NDB to indicate that it is currently active and running. The heartbeat is implemented by a NameNode first reading the highest counter value in a LEADER table and then updating its counter value in the LEADER table to be one higher than the current highest counter value. A heartbeat, in effect, increments the highest counter value. Heartbeats should be loosely synchronized by configuring hosts to use Network Time Protocol, and setting equivalent timeouts for triggering heartbeats at nodes. To ensure serialized updates to counter values, when each NameNode reads the highest counter value, it acquires a table-level write lock on the LEADER table and only releases this lock after it has updated its own counter value. All of these operations happen within a transaction, so that the lock is released in the event of a NameNode failure while updating the LEADER table.
Case-I: Failure-free scenario In the case of no failures, after each round when all NameNodes have successfully sent heartbeats, the highest value of the counter should have increased by the number of correct NameNodes, and the value of each NameNode's counter should have increased by approximately the number of correct NameNodes. We say approximately, as clock skew, network delays, and congestion may cause updates to arrive at varying times within a round. Assuming clock skews are not as large as the heartbeat interval and no congestion, all NameNodes should succeed in sending one heartbeat at least every 2 rounds. The above figure shows an example of counter values for 3 NameNodes with ids 1, 2 and 3 and their corresponding counter values as 23, 24 and 25 respectively. For brevity, we designate a NameNode with an id of value x as NNx. From this example we see that NN1 has the lowest id and is therefore elected as the current leader in the system.

Case-II: Leader crash scenario

For numerous reasons, a NameNode may fail in sending a heartbeat to NDB a round, and, thus, fail to update its counter value in the LEADER table. The counter value for the NameNodes remains the same Namenodes would experience an irregular sequence of counter values in each of these rows. For example, let’s say that NN1 has crashed and the current counter value is now 6. This would allow NN2 and NN3 to progress in updating the counter with values 9 and 10 while counter value for NN1 is still at value 6. In Figure 1.5, “Leader Table in NDB”, we can see a snapshot of the view on the LEADER table at this round (T0). The figure shows non-consecutive counter values 6, 9 and 10 when we would expect something like 8, 9 and 10. As all NNs have a consistent view of these counter values, so NN2 and NN3 can now agree that NN1 has crashed, and NN2 will become leader.

Figure 1.5. Leader Table in NDB

T_0

id	counter	timestamp	hostname
1	6	2012/10/19 22:01:10	cloud1
2	7	2012/10/19 22:01:10	cloud2
3	8	2012/10/19 22:01:11	cloud3

T_2

id	counter	timestamp	hostname
1	6	2012/10/19 22:01:10	cloud1
2	10	2012/10/19 22:01:13	cloud2
3	11	2012/10/19 22:01:13	cloud3

Determining the leader

The next step is to detect an irregular sequence of counter values and to decide if a new leader is to be elected. The basic idea to determine the leader is to determine which NameNode have counter values close enough to the current highest counter value to be considered 'correct'. In the example given above, the highest counter value at round T2 (bottom table) is 11. As we have 3 NameNodes in the system, we expect the counter values to be in the range [9-11], although allowing for out-of-order heartbeats, the counter values could be in the range [7-10]. The upper-bound on out-of-order updates to counters is, by default, 2 rounds. That means that a NameNode that is more than 2 rounds behind in updating its counter value is no longer considered to be correct. In the lower table, we can see that NN1 is now no longer considered correct, as both NN2 and NN3 have succeeded in updating their counters 3 times before NN1 has updated its counter. Therefore, NN2 now becomes the new leader, as its counter value is within the range of 2 rounds from the max counter value, and it has the lowest id of the remaining correct NNs. Both NN2 and NN3 independently reach a decision about the new leader, NN2, the next time they try to update their own counter.

Ensuring single leader dominance

Once a NameNode determines that it is the leader, the first thing it does is to ensure that there are no other leaders in the system. This means that all previously elected leaders (or NameNodes with ids lower than its own) have crashed. To ensure this, it enforces this rule by removing all records from the [LEADER] table for NNs with a lower id than its own.

Example 1.1. Leader Election Algorithm at NameNodes

```
function updateCounter()
retrieve ([COUNTER], counter)
increment counter
store([COUNTER], counter)

// The entry for this NN may not exist if it crashed
// and was removed by another leader
if(!exists([LEADER], id) then
id = retrieve([LEADER], max(id)) + 1
end if

store([LEADER], id, counter)

// The function that determines the current leader
function select()
SELECT id FROM [LEADER]
WHERE counter  (max(counter) – count(id)) // returns all correct NNs
LIMIT 1 /*selects lowest id*/
return id

// The function that returns the list of correct NNs
function selectAll()
SELECT id FROM [LEADER]
WHERE counter > (max(counter) – count(id)) // returns all correct NNs
ORDER BY id                              // the leader has the lowest id
return list(id)

upon event <init> do
leader = NIL
updateCounter()
leader = select()

// After every interval, the NameNode updates the counter
upon event <check>
updateCounter()
leader = select()
// If this NN is elected the leader, remove previous leaders
if(leader == id) then
remove([LEADER],[ids < leader])
end if

upon event < timeout>
// Kill NN on timeouts, so DNs can connect to the next leader NN
shutdown()

// Return the list of correct NNs in order of lowest ids
upon event <heartbeats>
return selectAll()
```

Example 1.2. Detecting the Leader at DataNodes

```
// The NN leader id is passed in upon initialization of DN
upon event init<list(id)>
nnlist = list(id)
// The NN with the lowest id is the leader
leader = min(nnlist)

// Update the list of NNs on every heartbeat response from Leader NN
upon event <heartbeat-response, list(id)>
nnlist = list(id)
leader = min(nnlist)

// On timeout from leader NN
upon event <timeout>
// remove current leader from nnlist
remove(nnlist, leader)
// elect a new leader
leader = min(nnlist)
```

All DataNodes are kept up-to-date with the current view of NameNodes in the system. This is done via requesting the NameNodes for the current list of NameNodes. This is done via a simple RPC call. The DataNodes get the list of NameNodes and assume the NameNode with the lowest id is the leader. When NN1 had crashed, the DNs keep retrying for some amount of time and if they are not successful at making contact with the NN it will remove it from the list and select the next NN who could potentially be the leader. This would achieve [Property#2]. There can be two possibilities where (a) the DataNode contacts the next NN and if it is actually the leader then the process flows normally. (b) The DataNode contacts the next NN in the list but who may not be the leader (because it is possible that it has also crashed or a new NN has joined with a lower id). This NN would then either provide the updated list of NameNodes, including the new leader, or not respond due to failure. THe DataNode can continue querying all NameNodes until either it is returned a list of correct nodes (including the id of the leader) or all NameNodes have failed. If there is a correct NameNode, eventually all correct DNs would recognize a correct NN as the leader thereby fulfilling [Property#1].

Case-II: Scenario when current leader thinks it is alive but cannot connect to NDB If NN1 is active and running, but it cannot update the LEADER table in NDB, then such a NameNode is not considered correct as per definition. In this scenario, all DataNodes would always think that NN1 is the current leader as it can make contact with that NN. But since the NN cannot update NDB, it has to kill itself and shutdown (or restart) hoping that some other NameNode would eventually be elected the leader. When NN1 is shutdown, DataNodes will keep retrying for some amount of time after some point where they will switch to the next NameNode and try to determine the next NN leader.

Ensuring Correctness of Hop HDFS

One of the main challenges in migrating HDFS' metadata from the heap of a single JVM to a relational database was ensuring the correctness of our approach. From a practical perspective, we ensure correctness by ensuring that we have (almost) no failing unit tests from the extensive suite of over 300 unit tests. From a theoretical perspective, our solution provides for support for multiple concurrent writers, so we need to show that our solution is free of both deadlock and livelock. Our first challenge was to migrate the state of HDFS from highly optimized data structures in the NameNode to tables in NDB. We migrated a total of 13 datastructures from HDFS' NameNode to tables in NDB, see Figure 1.6, “Entity Relation Diagram for NameNode state in Hop-HDFS”.

Figure 1.6. Entity Relation Diagram for NameNode state in Hop-HDFS

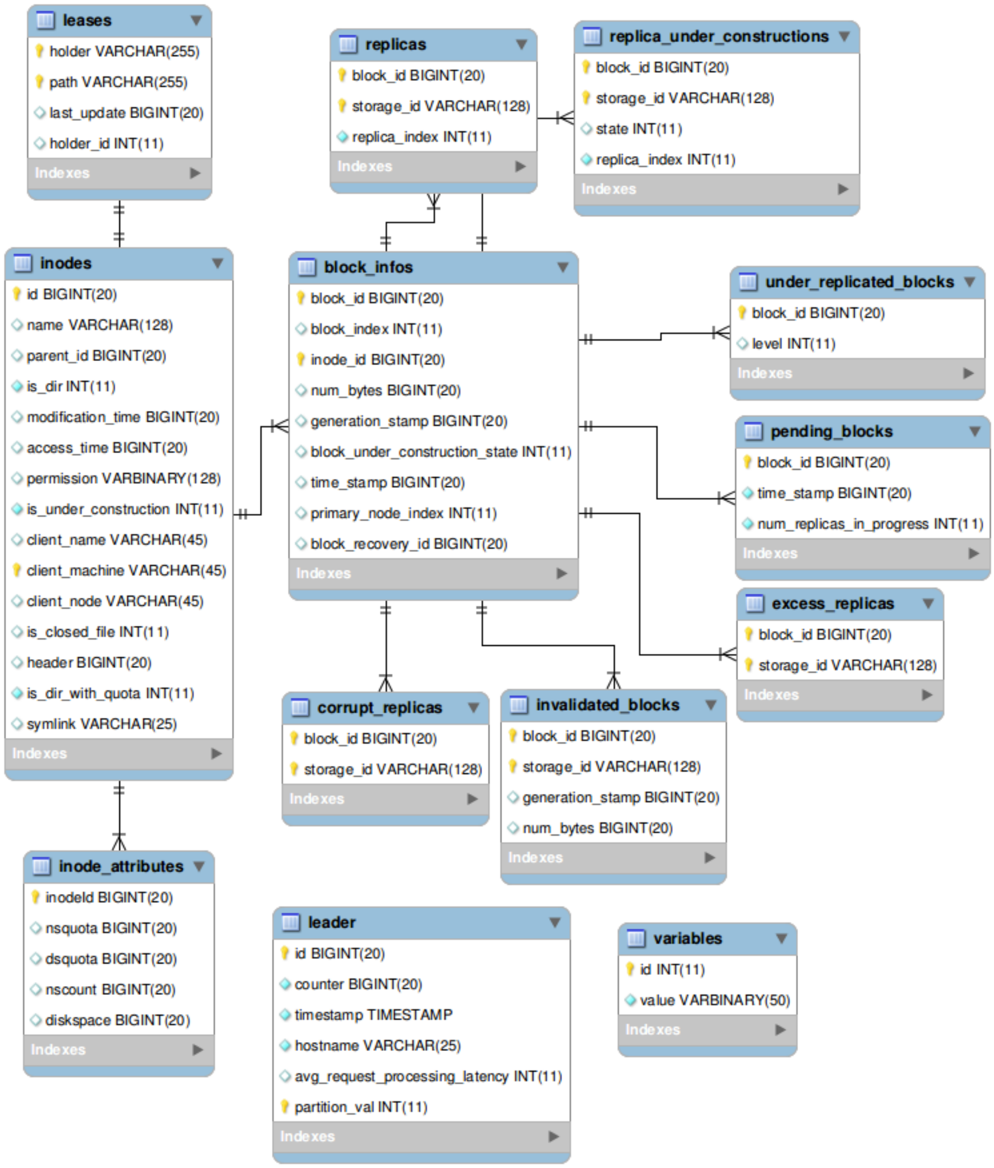
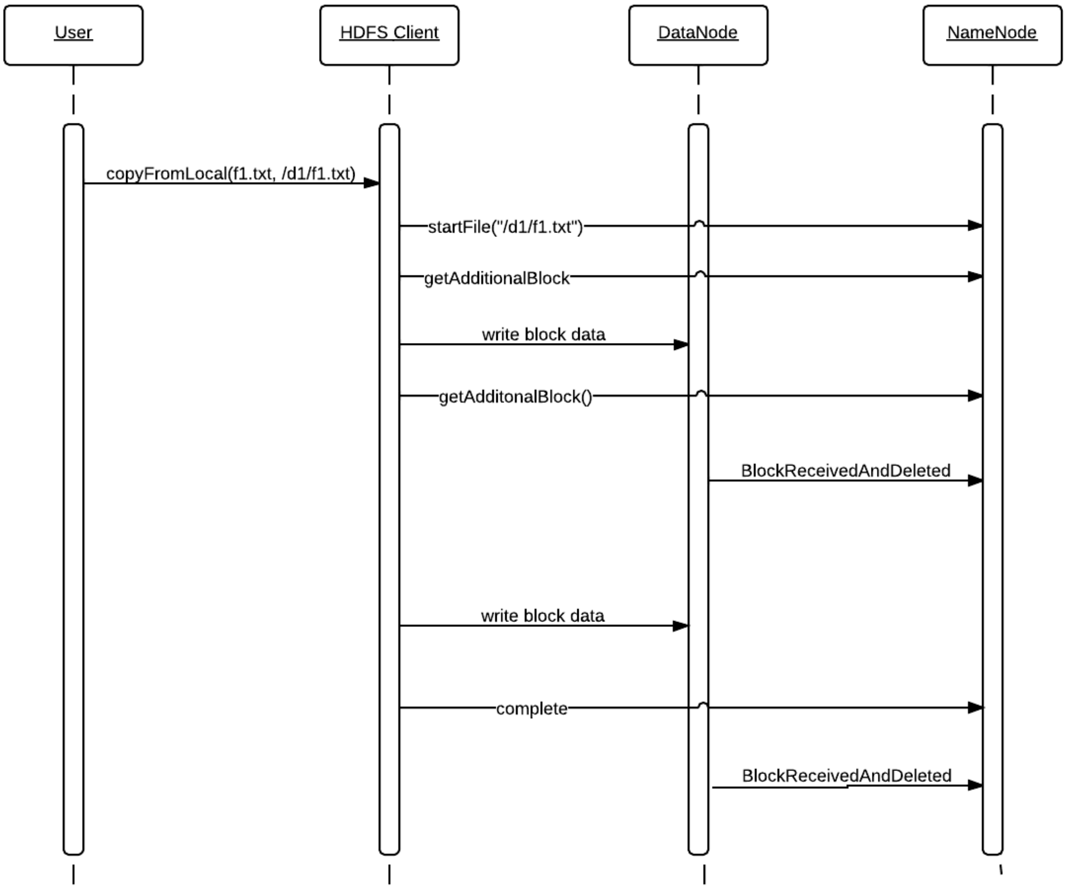


Figure 1.7. HDFS Filesystem Operation Sequence Diagram



Transactions and Isolation Levels

We implemented each HDFS operation as a single transaction, where we begin the transaction, read and write the necessary meta-data from NDB, and then either commit, or in the case of failure, abort the transaction and then possibly retry the transaction. An example of a HDFS filesystem operation that copies a file from the local filesystem into HDFS is ilustrated in Figure 1.7, “HDFS Filesystem Operation Sequence Diagram”. In this example, we can see that the copyFromLocal soperation consists of a number of HDFS internal protocol messages (startFile, getAdditionalBlock, writeBlockData, and getAdditionalBlock). Each protocol message sent to the NameNode is handled within a single transaction. Transactions, however, only maintain consistency for individual protocol messages, and HDFS maintains consistency for the filesystem between transactions using leases on files. A lease is held throughout the duration of the HDFS filesystem operation, and in the event of a failure, the lease will eventually be released after a relatively long timeout.

We implement our transactions using the MySQL CLuster database, also known as NDB (Network Database). Read-committed is the only isolation level supported by NDB, and is a general, cheap, and widely supported transaction isolation level. Read-committed isolation means a transaction may see new state from other transactions that have completed while that transaction is executing. The read-committed isolation level allows anomalies, such as fuzzy reads, phantom reads, lost update, read skew and write skew [2], that would break the consistency of HDFS' file-system operations.

Figure 1.8. Locking in conflicting order can lead to Deadlock

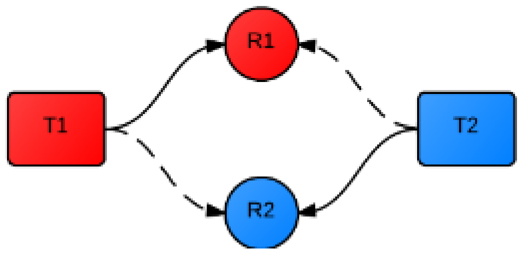
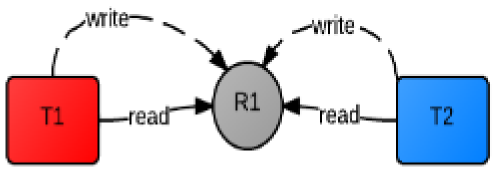


Figure 1.9. Lock upgrade can lead to Deadlock



Hierarchy of Resources

Filesystems, such as HDFS, have hierarchical namespaces, with directories containing zero to many files. In unix-based filesystems, both files and directories are represented as inodes. A filesystem hierarchy can be represented as a directed acyclic graph (DAG), when symbolic links are resolved. We ensure deadlock freedom, by ensuring that all transactions lock inodes in a consistent order, see Figure 1.8, “Locking in conflicting order can lead to Deadlock”. The consistent order is defined by the filesystem's DAG: the appropriate read or write locks are taken at inodes, starting at the root and traversing directories in a depth-first order to the leftmost leaf info, continuing until either the rightmost leaf inode is reached or all the inode locks have been acquired for this transaction. The locks are then held for the duration of the transaction. Locks are not upgraded within a transaction. Since all transactions acquire locks in the same order, there are no cycles, and since locks are not upgraded, there will be no lock-upgrade deadlocks, , see Figure 1.9, “Lock upgrade can lead to Deadlock”. Locks also need to be acquired on data-structures used by inodes, such as blocks, replicas, leases, corrupted-blocks, etc. In HDFS, an inode for a file contains a variable number of fix-sized blocks (typically 512 MB), where each block is typically replicated on 3 DataNodes. When operations are performed on inodes or a constituent part of an inode, we first take a write lock on the inode itself, ensuring no other transactions can concurrently access that inode or its constituent parts.

Snapshot layer

To improve the performance of our solution, we take snapshots of the database state at the start of filesystem operations, mutate that state locally, and then finally commit any changes at the end of the transaction. This approach reduces the number of round-trips to the database, helping to improve database throughput.

Example 1.3. Snapshotting at NameNodes

```
init: snapshot.clear

operation doOperation
    tx.begin
    doSnapshot()
    performTask()
    tx.commit

operation doSnapshot
    S = total_order_sort(op.X)
    foreach x in S do
        if x is a parent then level = x.parent_level_lock
        else level = x.strongest_lock_type
        tx.lockLevel(level)
        snapshot <-tx.find(x.query)

operation performTask
    //Operation Body, accessing only state stored in the snapshot
```

For some filesystem operations, the presented algorithm is not enough to ensure a total order on acquiring resources. For example there might be a datatype y for which x is parent, and a lock on x should be acquired to access y. Since y is not supplied in the operations' parameters we need to first discover y before we can lock it. Only then, can we start the main transaction by locking y before locking x. We solve this problem by breaking the operation up into two transactions. In the first transaction we do not mutate state, we only resolve resources that need to be locked in the second transaction. In the second transaction, we first validate that the resources/state we resolved in the first transaction is still valid, and if so, we then perform a standard transaction, acquiring locks and taking a snapshot of the data. If the state in the second transaction is no longer valid, we can restart the operation, beginning at the first transaction.

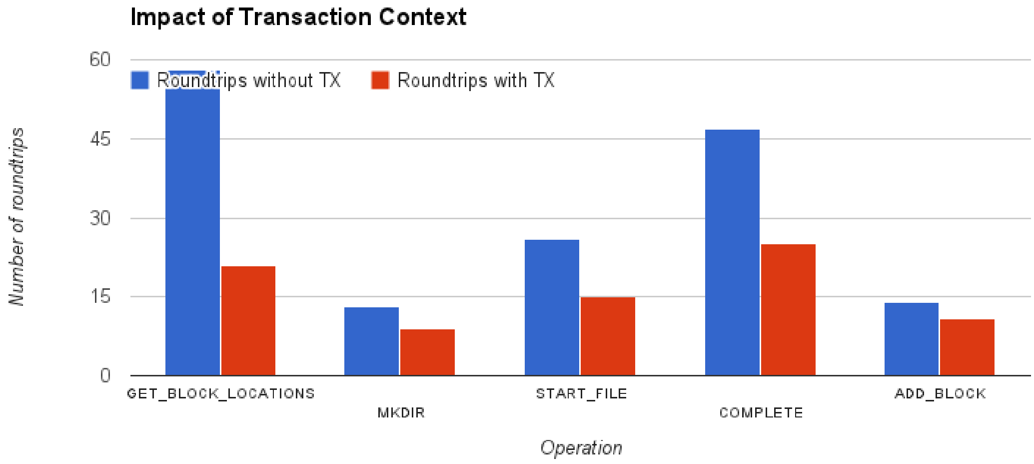
Example 1.4. Snapshotting at NameNodes using Two Transactions

```
init: snapshot.clear, restart = true, try = 0

operation doOperation
    while restart and try < 3
        restart = false
        try = try + 1
        if op.should_resolve_parent then
            tx1.begin
            resolve_parent(op.param)
            tx1.commit

        tx2.begin
        doSnapshot()
        if data from tx1 is not valid then
            tx.abort()
            restart = true
        else
            performTask()
            tx2.commit
    end while

operation resolve_parent(y)
    tx.lockLevel("read_commited")
    tx.find(y.parent_query)

operation doSnapshot
 S = total_order_sort(op.X)
 foreach x in S do
  if x is a parent then level = x.parent_level_lock
                   else level = x.strongest_lock_type
  tx.lockLevel(level)
  cache <- tx.find(x.query)

operation performTask
 //Operation Body, accessing only state stored in the snapshot
```

Performance of Hop HDFS

We have recently re-written our code base to migrate to HDFS 2.2, which is functionally complete, but it is too early to give read/write performance figures for our new implementation. Unpublished performance figures from an earlier prototype show that Hop-HDFS can scale to handle a similar number of read and write requests per unit time as Apache HDFS, but needing additional hardware to do so. We have introduced a number of features to enable this high level of performance, including a snapshot layer at NameNodes and row-level locking at the database level, rather than a system level lock for update operations as is done in Apache HDFS. Our snapshotting layer involves a transaction acquiring all resources it requires at the start of a primitive filesystem operation, and performing local read/write operations on the snapshot copy, and then finally committing or rolling back on transaction commit.

Figure 1.10. Reduction in the number of DB roundtrips by snapshotting



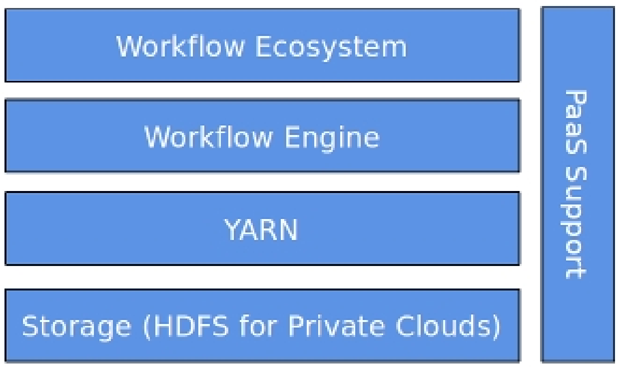
For improved performance, we have also implemented schema-aware partitioning of metadata across multiple hosts, ensuring all data related to a single inode hashes to the same host, and can be retrieved in a single network hop. This involved partitioning tables by inode-id, and when a transaction is started that involves a single inode, we give a hint to NDB, using the ClusterJ API, that the transaction should start on data node containing the data for that inode.

One positive aspect of HopHDFS is that the amount of metadata that can be supported is vastly increased over Apache HDFS, and it is much more easy to modify. In Apache HDFS, the NameNode is limited to the maximum size of the JVM heap for a single node, which in practice is around 100GB. In NDB, there are existing clusters of 2TB, and larger clusters are feasible. However, we have measured an expansion in the amount of memory that we require of around 43 percent. Our plan is to use this extra metadata for features such as block-level indexing, which will be very useful for BAM files used to store whole sequenced genomes, and also access control information for files. The lack of access control in HDFS is one enterprise-level feature that is preventing the use of HDFS in storing sensitive data, such as genomic data.

Hop: Platform-as-a-Service support for HDFS

HDFS exists as just one part of the larger Hadoop ecosystem for the storage and parallel processing of big data. As you can see in Figure 1.11, “Hop architecture”, our goal is to build an architecture where HDFS is the main layer for storage, YARN is used to manage the allocation of computational resources, and we support a workflow manager and workflow language for Bioinformatics (Cuneform) as data-intensive computing support for BiobankCloud. Platform-as-a-Service (Paas) support is required at all layers in our system. Security and data sharing are not covered in this diagram, as they will be integrated in future phases of the project.

Figure 1.11. Hop architecture



.

1. *YARN* Yarn (Yet Another Resource Negotiator) manages the allocation of computation and memory resources to tasks in Hadoop clusters [1]. It supports many processing models, not just Map-Reduce, by separating the old JobTracker into a Resource Manager, a Scheduler and Application Master. The Application Master has the flexibility to accommodate heterogeneous processes by implementing a wrapper for each kind of application that runs over YARN, so that the application itself manages processing resources allocated to it. This enables the user to process data intensive task like MapReduce jobs or, in the BiobankCloud, to run a bioinformatics workflow engine that makes use of YARN to handle and negotiate the scheduling of its jobs.

2. *Workflow Engine* On top of YARN, the BiobankCloud workflow engine parses bioinformatic workflows written in Cuneform into an execution model of arbitrary tasks. For each task, it asks YARN for at least one container, then for each container allocated task based on the scheduling policy it stages in data into HDFS, launches the task and stages out the result, if needed. The workflow engine and language are being developed by Humboldt University as part of the BiobankCloud project.

Hop PaaS

Our PaaS for HDFS, as well as YARN and the Workflow Engine, supports the automated deployment of a Hop cluster on Amazon Cloud, Open Stack or a bare metal cluster. We call it Hadoop Open Platform-as-a-Service, or *Hop* for short, as it is *open* - designed to be deployable on any cloud platform or a bare-metal environment, not just those we currently support. Hop consists of a set of frameworks and libraries that we use to support the automated deploymented a Hadoop cluster from a website or the command-line. The main technologies we have built our prototype PaaS on are Chef and JClouds, but we are unifying all these technologies in an API based on a YAML that can be used to define a cluster that is to be deployed, see Figure 1.12, “Deploying a Hop cluster from a Website”. The other technologies we currently use are BitTorrent for improving the download speeds of installation binaries and virtual machine instances (AMIs in AWS) to pre-load software onto virtual machines. The YAML API can currently be accessed by a Dashboard application that we have built to manage and adminster Hop clusters.

Figure 1.12. Deploying a Hop cluster from a Website

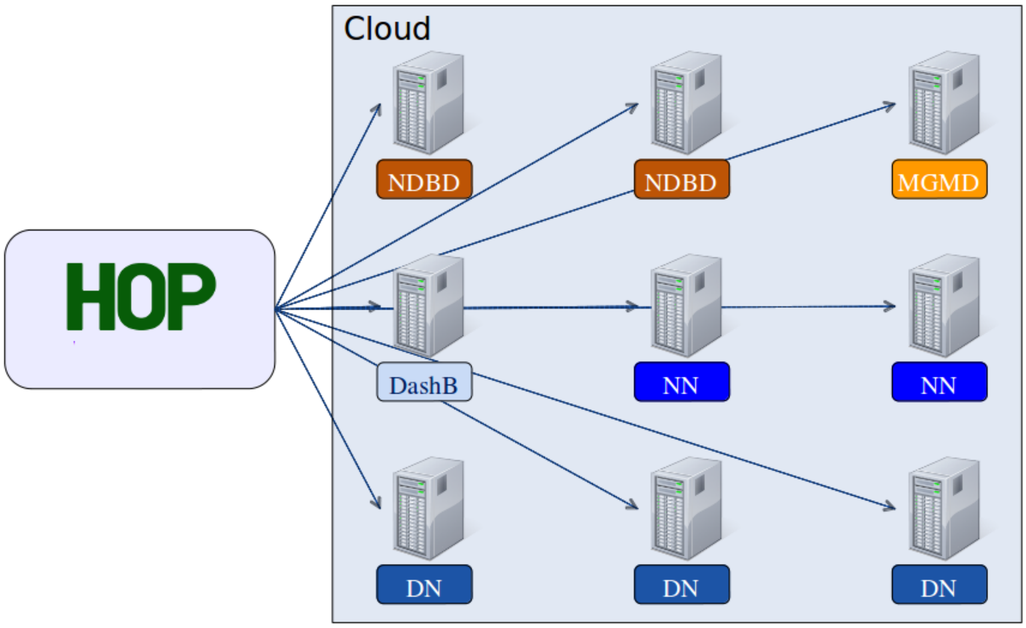
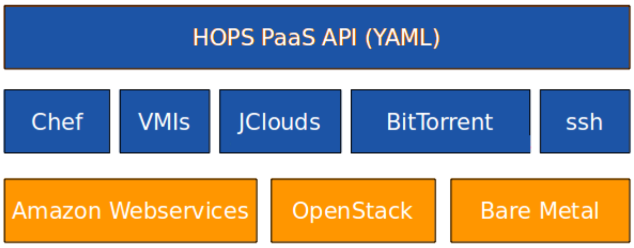


Figure 1.13. Hop PaaS API



1. *YAML* YAML (YAML Ain't Markup Language) is markup language which takes concepts from programming languages such as C, Perl and Python, and ideas from XML. We use YAML to define the set of hosts and the services that will be installed on those hosts, making up a single cluster. This way, we can define a whole cluster in a single file, enabling easier management of clusters and even the sharing of cluster definitions. YAML syntax allows easy mappings of common data types found in high level languages like list, associative arrays and scalar. It makes it suitable for tasks where humans are likely to view or edit data structures, such as configuration files or in our case, cluster definition files. Additionally, we make use of the open source parser SnakeYAML to parse the contents of our cluster definition files. The SnakeYAML parser transforms the given cluster definition into consecutive stages such as defining security groups, virtual machine allocation, bittorent, installation, validation and retry. An example of a simple cluster definition is given in Example 1.5, “Example Cluster Definition”. The YAML file defines a cluster consisting of 6 nodes, with MySQL Cluster running exclusively on 2 nodes, Hadoop running exclusively on 3 nodes, and another node running both MySQL Cluster and Hadoop services. The example uses default values for the AWS image, instanceType and region. There are many other parameters that can be overriden. Our services map directly onto chef recipes for installing the services. We are developing a model for explicitly handling dependencies in chef, so that dependent services such as Java don't need to be specified as requirements in this cluster definition file.

   #### Example 1.5. Example Cluster Definition

   ```
   name: simpleCluster
   provider:
   name: aws-ec2

   nodes:
   - service:
   - ndb::ndbd
   number: 2

   - service:
   - ndb::mgm, ndb::mysqld, hadoop::namenode
   number: 1

   - service:
   - hadoop::namenode
   - hadoop::resourcemanager
   number: 1

   - service:
   - hadoop::datanode
   - hadoop::nodemanager
   number: 2
   ```

2. *Apache JClouds* Apache JClouds is an open source multi-cloud api interface which allows us to write reusable code for creating, destroying, and bootstrapping virtual machines (VMs) on different

cloud providers. The same code can be configured to interact with Amazon, OpenStack, Azure, and Rackspace VMs, and 26 other cloud providers. Through JClouds simple Java interface, we can deploy and port Hop to different cloud environments. Hop parses cluster definition files, producing code that executes JCloud API calls to create, destroy and bootstrap VMs.

3. *Chef* Chef is a systems infrastructure and configuration framework that automates the deployment of servers and applications to any physical, virtual or cloud location. JClouds is used to bootstrap chef on VMs or physical hosts, and once chef is installed on a host, we can run *chef recipes* using the chef-client deployed on host to install software on that host. The chef-client relies in a series of abstract definitions (defined as cookbooks and recopes) which are managed in Ruby and are treated like source code. With each definition, we describe how a specific part should be built and managed, which then; the chef-client applies these definitions to deploy and configure servers and applications as specified. In most of the cases, it is simple enough to let chef-client know which cookbooks and recipes it needs to apply.

4. *BitTorrent* After machines are allocated in cloud, with the metadata information that JCloud returns, dashboard tries to open a ssh connection into every single machine and install Chef agent for installations. Before installation starts, software libraries is replicated in all machines from dashboard, though the process could overflow the bandwidth to dashboard if all machines try to download from dashboard. To handle this situation HopS run a bittorent in which dashboard machine is the seeder, then all machnies could contribute to download process which is both faster and anti-bottleneck. After download Chef agent starts installation based on the required packages in each machine and with the order of dependencies between packages.

References

[1] Vinod Kumar Vavilapalli, Arun C Murthy, Chris Douglas, Sharad Agarwal, Mahadev Konar, Robert Evans, Thomas Graves, Jason Lowe, Hitesh Shah, Siddharth Seth, Bikas Saha, Carlo Curino, Owen O'Malley, Sanjay Radia, Benjamin Reed, and Eric Baldeschwieler. "Apache Hadoop YARN: Yet Another Resource Negotiator". *2013 ACM Symposium on Cloud Computing (SoCC 2013)*. 2013. http://hortonworks.com/blog/apache-hadoop-yarn-wins-best-paper-award-at-socc-2013/ .

[2] Hal Berenson, Phil Bernstein, Jim Gray, Jim Melton, Elizabeth O'Neil, and Patrick O'Neil. "A critique of ANSI SQL isolation levels". *ACM SIGMOD Record*. ACM. 24. 1–10. 1995.

[3] D Borthakur. "The hadoop distributed file system: Architecture and design". *Hadoop Project Website*. 1–14. 2007. http://cloudcomputing.googlecode.com/svn/trunk/??/Hadoop_0.18_doc/hdfs_design.pdf .

[4] Dhruba Borthakur, Samuel Rash, Rodrigo Schmidt, Amitanand Aiyer, Jonathan Gray, Joydeep Sen Sarma, Kannan Muthukkaruppan, Nicolas Spiegelberg, Hairong Kuang, Karthik Ranganathan, Dmytro Molkov, and Aravind Menon. "Apache hadoop goes realtime at Facebook". 1071. 2011. http://dl.acm.org/citation.cfm?id=1989323.1989438 .

[5] Dennis Fetterly, Maya Haridasan, Michael Isard, and Swaminathan Sundararaman. "TidyFS: a simple and small distributed file system". 34. 2011. http://dl.acm.org/citation.cfm?id=2002181.2002215 .

[6] Seth Gilbert and Nancy Lynch. "Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services". 33. 51. 2002.

[7] Howard Gobioff, Sanjay Ghemawat, and Shun-Tak Leung. "The Google file system". 37. 29. 2003.

[8] Haryadi S Gunawi, Abhishek Rajimwale, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. "SQCK: A Declarative File System Checker.". 131–146. 2008.

[9] J. Gray, R. Lorie, G. Putzolu and I. Traiger. "Granularity of Locks and Degrees of Consistency in a Shared Database. In Modeling in Data Base Management Systems". 365–394. 1976.

[10] Leslie Lamport. "The part-time parliament". *ACM Transactions on Computer Systems (TOCS)*. ACM. 16. 133–169. 1998.

[11] Mysql Reference Manual. “MySQL 5.0 Reference Manual”. *Syntax*. 3079. 2010.

[12] Lev Novik, Irena Hudis, Douglas B Terry, Sanjay Anand, Vivek Jhaveri, Ashish Shah, and Yunxin Wu. “Peer-to-peer replication in WinFS”. *Technical ReportMSR-TR-2006-78, Microsoft Research*. 2006.

[13] Konstantin V Shvachko. “HDFS Scalability: The limits to growth”. *login*. 35. 6–16. 2010.

[14] Feng Wang, Jie Qiu, Jie Yang, Bo Dong, Xinhui Li, and Ying Li. “Hadoop high availability through metadata replication”. 37–44. 2009.

Chapter 2. Quickstart with Vagrant

This section describes the steps required to deploy a Hop cluster on a single machine using git, vagrant [1] , and chef [2] .

Pre-requisites:

You should have the following programs installed: git and vagrant. You will also need to download the vagrant virtual machine image for Ubuntu 12.04 "precise".

```
apt-get install git-core vagrant
vagrant box add "precise64" http://files.vagrantup.com/precise64.box
```

Launching Vagrant

You are ready to clone the chef recipes, and launch a vagrant instance.

```
git clone https://github.com/hopstart/hop-chef.git
cd hop-chef
vagrant up
```

Now grab a coffee, assuming you have a good network connection, it will take around 15 minutes to provision a vagrant instance. When vagrant successfully completes provisioning using chef, use the following URL and default user credentials to access the Hop Dashboard:

```
https://localhost:9191/hops-dashboard/
user: admin
password: admin
```

You can log into the VM and then get root access using:

```
vagrant ssh
sudo su
```

If needed, you can configure the glassfish webserver here:

```
https://localhost:5858
user: admin
password: admin
```

You can now jump to Chapter 4, *Hop Dashboard*

[1] Vagrant is a tool for building complete development environments. With an easy-to-use workflow and focus on automation, Vagrant lowers development environment setup time, increases development/production parity, and makes the "works on my machine" excuse a relic of the past. [http://www.vagrantup.com]
[2] Chef is a systems and cloud infrastructure automation framework that makes it easy to deploy servers and applications to any physical, virtual, or cloud location, no matter the size of the infrastructure. [http://docs.opscode.com/]

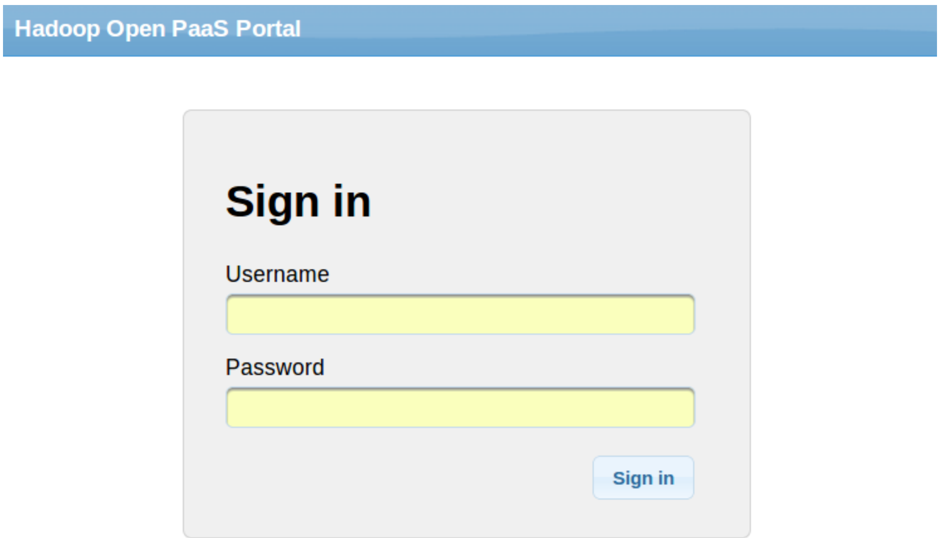
Chapter 3. Hop Web Portal

Hop is an effort to offer a high performance next generation hadoop platform focusing on high scalability, availability and reliability. This section describes the information needed to jump start Hop cluster. Hop web portal, is an entry point were users can test our platform in one of the supported environments: AWS, OpenStack and Baremetal machines. It is a simple web application that deploys the Hop Dashboard in the cloud. Using the Hop dashboard you can provision and configure the rest of the cluster. Hop Dashboard provides a centralized end-to-end management and monitoring application for Hop distribution. Following are the step by step instructions for setting up the cluster in different environments.

Example 3.1. HOP Portal

```
https://snurran.sics.se:8181/hop-portal
```

Figure 3.1. Hop Portal



Requirements:

Hop Dashboard has to be installed on a machine in a cluster that has connectivity both to cloud infrastructure and the cluster. Using Hop Web Portal you can provision a cluster and install the dashboard to any of the following plateforms: AWS, OpenStack and cluster of bare-metal hosts. The following environments are currently fully supported:

- *Amazon Web Services:* In order to use of our platform in AWS cloud Infrastructure, it is necessary to provide EC2 account credentials. When deploying our Dashboard in EC2, we recommend minimum Instance type should be of m1.large type Instance with Ubuntu.
- *OpenStack* For using Hop on OpenStack, it is necessary that you provide the credentials and configuration parameters to connect to you OpenStack end-point. The recommended instance to use in OpenStack infrastructure should be equivalent to a m1.large instance type or greater.
- *Baremetal Physical Machine:* For deployment on Baremetal machines our system requires security credentials of a user with sudo access to the machines to deploy the software through SSH .
- *Vagrant:* With our vagrant distribution, you can deploy the whole cluster locally in a VM machine. It is ideal for testing the platform before deployment in a production environment.

  All the environments are tested with instances running Ubuntu 12.04.

Installing the Hop Dashboard

Intalling Hop Dashboard through the Portal is quite simple and takes a couple of minutes to deploy. Here you may find instructions on how to deploy the dashboard on AWS, OpenStack and Baremetal Machine:

1. *Amazon EC2:* For installing the dashboard in Amazon EC2 follow these simple steps:

   - Login into the HOP Portal with your user name and password.
   - Select from the providers option in the maintoolbar, the Amazon EC2 option. Enter the following information in the new form

     - Dashboard credentials: admin's username and password in order to access your newly created dashboard.
     - EC2 credentials which include the Access Key id from you AWS account with its related Secret key.
     - Instance configuration parameters used to deploy a virtual machine in AWS.

       a. Security group where the machine will be deployed. If it does not exist, then a new security group will be created automatically.

       b. The hardware ID of the instance type we want to use from Amazon EC2. For example, m1.small, t1.micro. The recommended instance type is m1.large.

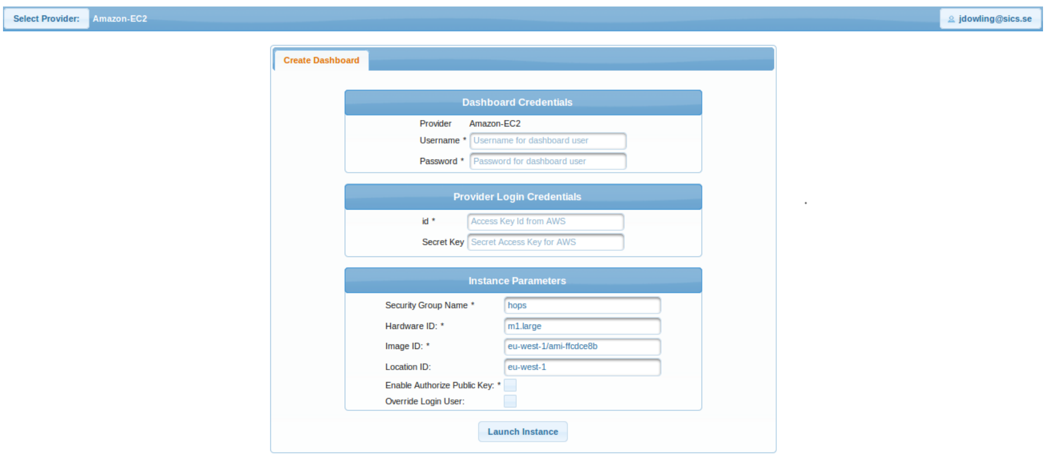
       c. Image ID which includes the region of that image and the *ami id* tag. We only support Ubuntu based images.

       d. Location ID of the region you want to deploy the dashboard.

       e. Selecting the option to authorize the public key will open a new option dialog box were you can insert your public key. By default we generate random key pairs for the machines through EC2 key pair service, and it is not possible to access the machines internally without this option.

       f. Selecting the override login user, will override the default user for Ubuntu AMI images with the login user of your choice. This is necessary if you use custom Ubuntu images which are not one of the Ubuntu images that canonical offer in AWS by default.

Figure 3.2. Portal AWS



- After filling up the form, press the Launch Instance button. The whole process takes 10-15 minutes. After the deployment you will receive a notification showing the URL of the newly deployed Hop Dashboard. To login the dashboard, use the credentials you specified previously in the web portal.

2. *OpenStack:* For deployment in OpenStack cloud follow these simple steps. Note, this is in alpha state:

   - Login into the Hop Portal with your user name and password.

   - Select from the providers the OpenStack option. A new form will be generated.

     - Dashboard credentials: here you specify the admin username and password for the new dashboard.

     - OpenStack credentials: the user name and password to access the OpenStack project. The username should be a concatenation of the OpenStack project name and the user for that project. For example "projectName:user". Also you should indicate the url of your OpenStack Nova end-point in order to send the requests to your OpenStack infrastructure.

     - Configuration parameters that are used to deploy a virtual machine in OpenStack:

       a. Security group where the machine will be deployed. If it does not exist, we will automatically create a security group and open the ports needed for the application.

       b. The hardware ID of the instance type we want to commission in OpenStack cloud. This is a number which corresponds to the type of instance you want to deploy and is supported by your OpenStack infrastructure. We recommended using a configuration similar to a m1.large in EC2.

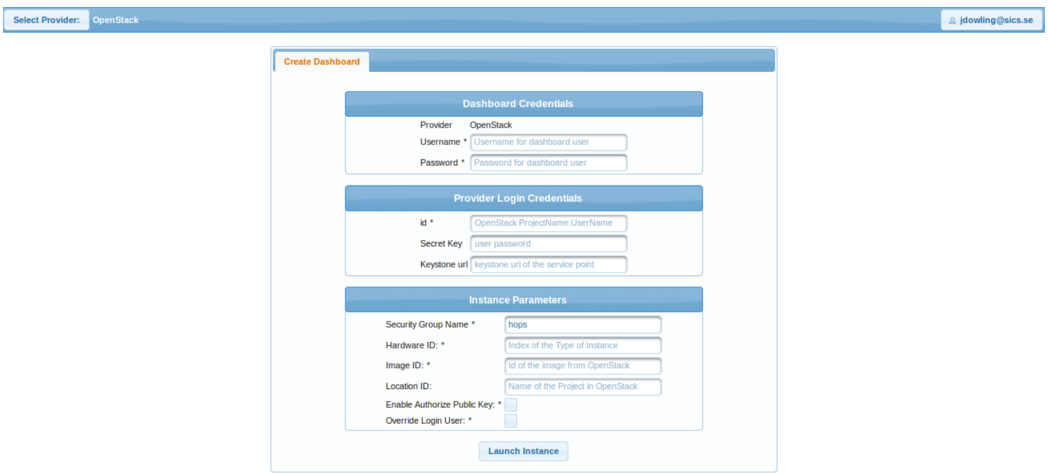
       c. An Image ID image located in the openstack project.

       d. Location ID identifies the dashboard in the OpenStack cluster.

       e. Selecting this option to authorize the public key based access. It will open a new dialog box were you can insert your desired public key. By default we generate random key pairs for the machine through OpenStack key pair service, and it is not possible to access the machine internally without selecting this option.

       f. Selecting the override login user: This is necessary for OpenStack if you are using a custom Ubuntu image.

Figure 3.3. Portal OpenStack



- After filling up the form, press the Launch Instance button. The whole process takes 10-15 minutes. After the deployment you will receive a notification showing the URL of the newly deployed Hop Dashboard. To login the dashboard, use the credentials you specified previously in the web portal

IP pools in OpenStack

It is necessary that you have allocated at least 1 public IP to the project. During the deployment phase the portal will query the OpenStack project and link the public ip to the VM.

3. *Baremetal Physical Machine:* For deploying the dashboard on a BareMetal hosts cluster follow these simple steps:

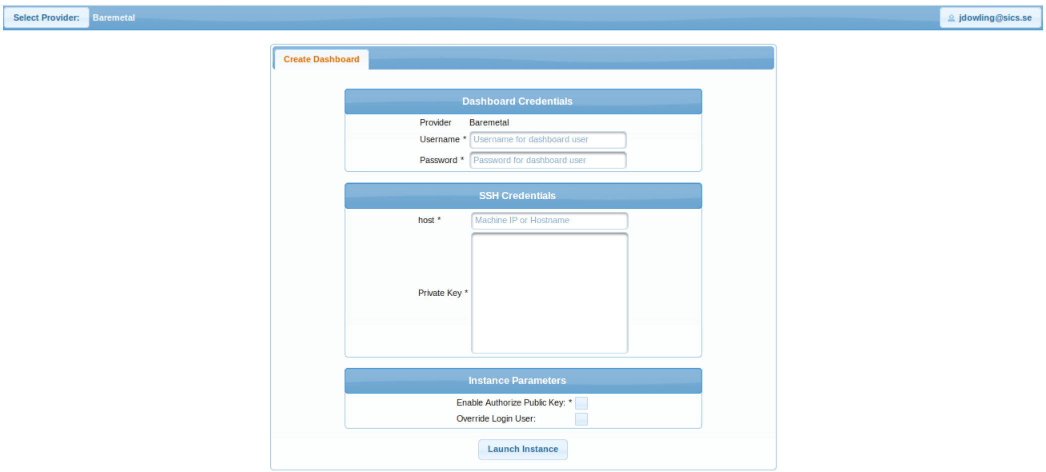
    - Login into the Hop Portal with your user name and password.
    - In the new page, select BareMetal from the providers list. A form will be generated were you need to fill in the following

        - Dashboard credentials: here you specify the admin username and password for the new dashboard.
        - SSH credentials: includes the host address of the machine we want to connect to and the private key.
        - Extra parameters that we might need:

            a. Selecting the option to authorize the public key, it will open a new option were you can insert your desired public key to allow extra access to the machine.

            b. Selecting the override login user will rename the sudo user to used to deploy the dashboard on the machine.

Figure 3.4. Portal Baremetal



- After filling up the form, press the Launch Instance button. The whole process takes 10-15 minutes. After the deployment you will receive a notification showing the URL of the newly deployed Hop Dashboard. To login the dashboard, use the credentials you specified previously in the web portal.

Chapter 4. Hop Dashboard

Hop Dashboard is a web application designed with Hadoop administrators in mind. The Dashboard provides a centralized end-to-end management and monitoring application for Hop clusters. It simplifies the Hadoop administrator's job by having all the necessary statistics of the Hadoop cluster gathered in one place and presented in an effective manner, and it also provides buttons to execute maintanance commands as well as command-line terminals for services like MySQL and HDFS. The Dashboard monitoring services are built using the open-source collectd monitoring software.

You will need to configure certain parameters when the Hop Dashboard is launched for the first time. In order to access the dashboard, use the credentials you specified in the Hop Portal when you where preparing to launch the dashboard Chapter 3, *Hop Web Portal*. After login, if you press your user icon you will have a menu with the following options:

- Change password
- Edit Graphs
- Backup/Restore
- Setup Credentials

We will go these options in more detail in the following sections.

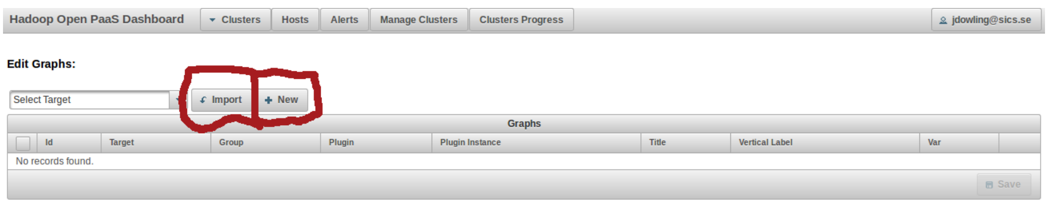
Change Password

Change the password for this user.

Edit Graphs

The Hop Dashboard enables the customization of what monitoring data is displayed to administrators. An administrator can select which graphs to enable for the different services of Hop.

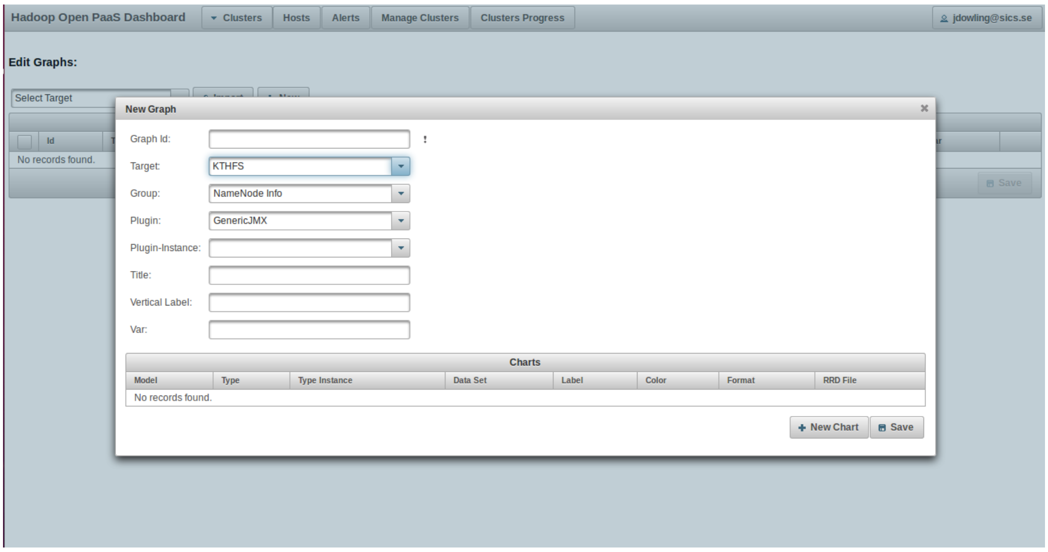
Figure 4.1. New/Import Graphs



You can define your graphs through two different options that are visible on this new view:

- *New* Selecting this option will open a new dialog where you can define the specifications for a new graph for statistics monitored by the dashboard.

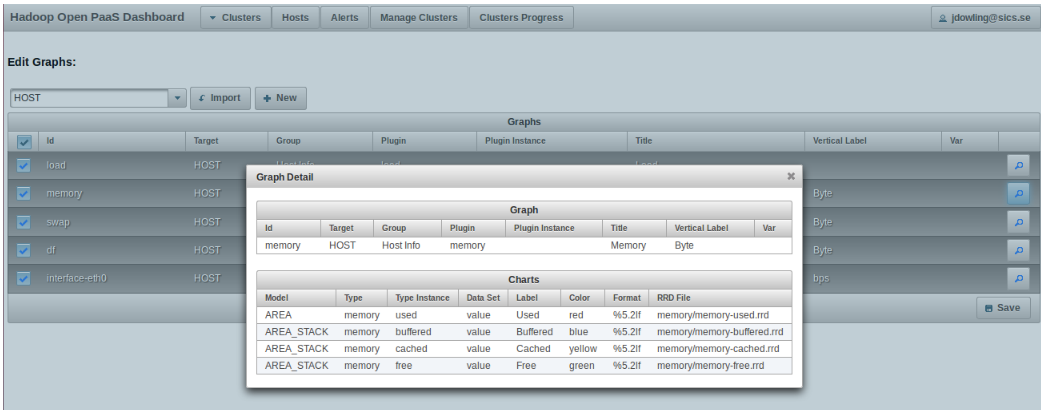
Figure 4.2. Graph Editor



- *Import* It is also possible to import a predefined set of graphs from a JSON file. This is the quickest way to load a large number of graphs.

The graph speficiations are stored in a MySQL database and loaded in the dashboard. In the graph table, it is possible to view all the graphs for a specific type of service or component that the dashboard is monitoring. Pressing the zoom button will generate a dialog box where you can see more details about the graph.

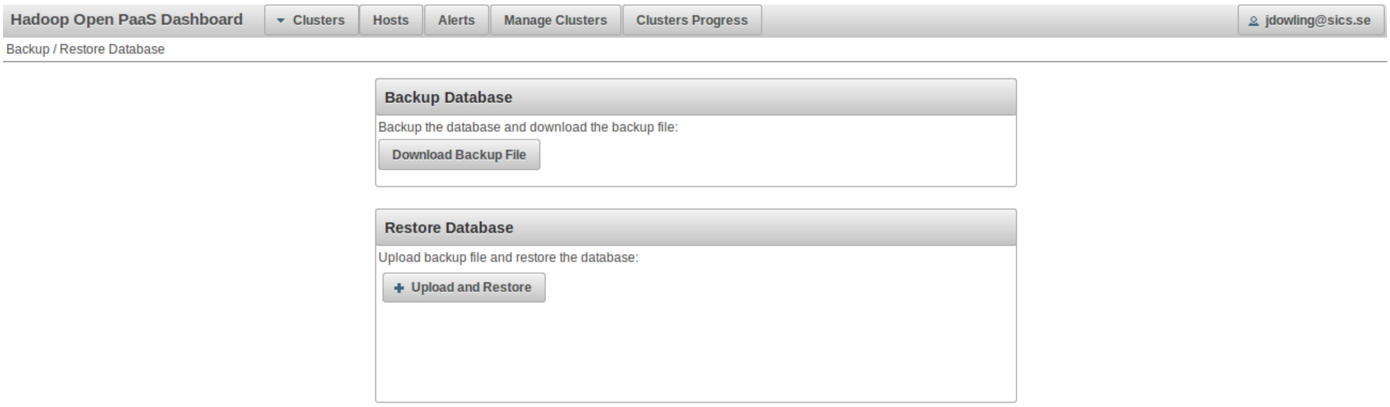
Figure 4.3. Graph Selection Detail



Backup/Restore

The Dashboard's state is stored entirely in a MySQL database instance. This option makes a backup of the whole dashboard state as an SQL file that can be downloaded by the user for off-site backup. In the case of failure or maintenence, the dashboard can restore a previous state from a backup SQL file.

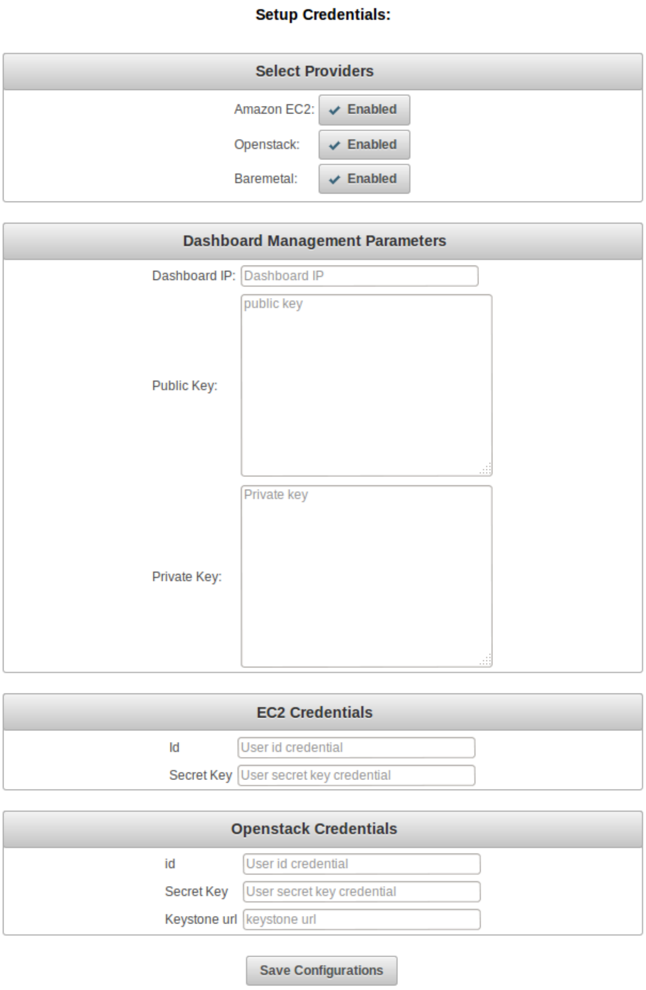
Figure 4.4. Backup/Restore



Setup Credentials

This option is to setup user credentials that are used when the dashboard deploys a cluster on a cloud platform or cluster. They are typically AWS or OpenStack credentials or a public key for cluster deployments. If user credentials have not been setup, cluster deployment will fail.

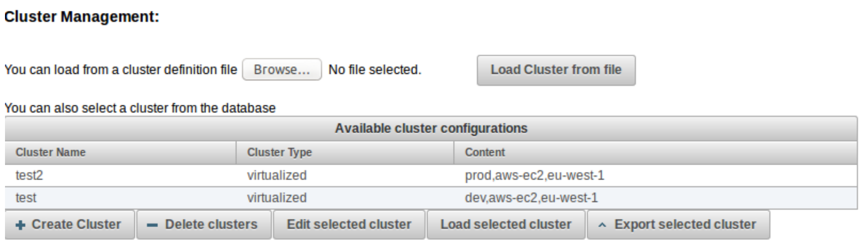
Figure 4.5. Setup Credentials



Cluster Management

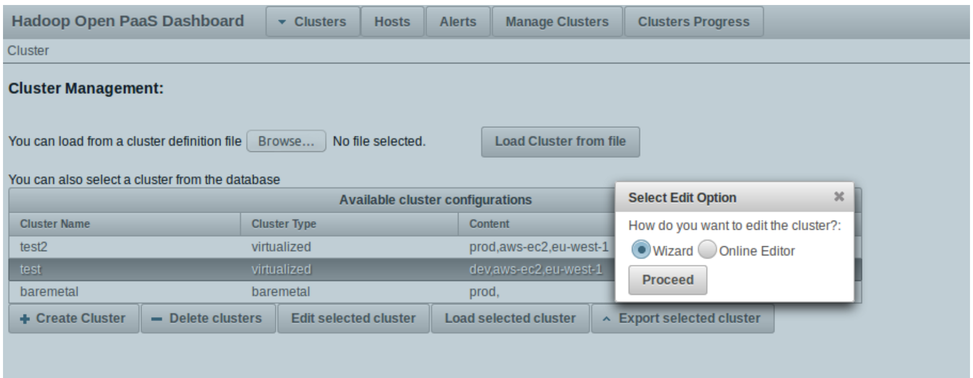
Cluster mangement is an interesting feature in the Hop Dashboard. The dashboad keeps track of diffenent cluster applications deployed in the cloud. It allows the administrators to create, delete, edit, load and export clusters.

Figure 4.6. Manage Cluster



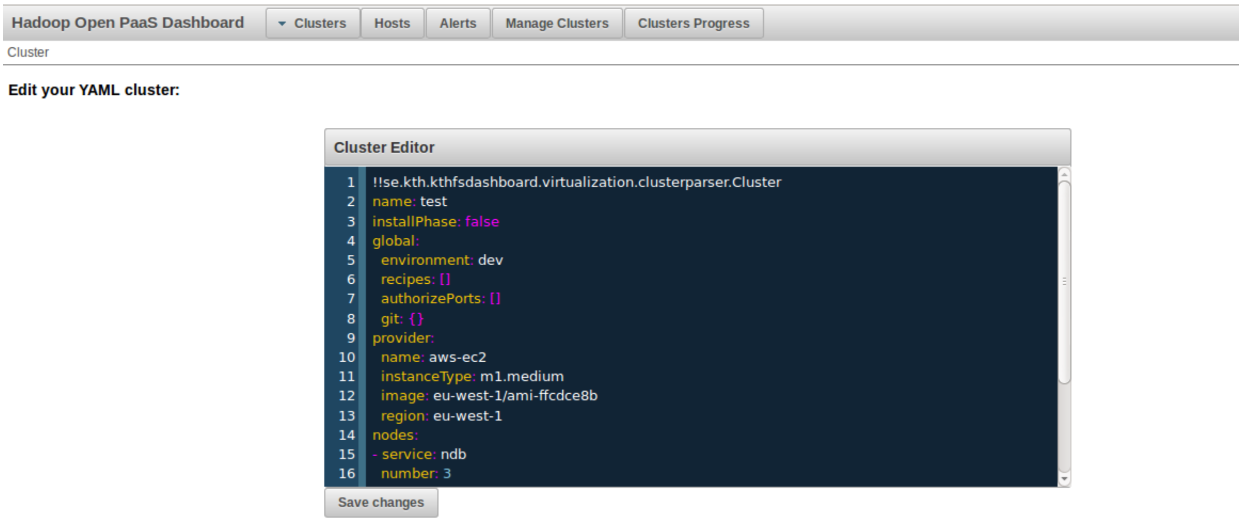
- *Create Cluster* Selecting this option will take you to the cluster generation wizard where a users can of generate their own clusters for Hop. In the next chapter, we will explain in detail how users can create their custom clusters using our Cluster Definition Language, see Chapter 5, *Defining a Cluster*.
- *Delete Clusters* This option will delete a selected cluster. It is possible to delete multiple clusters at once by shift clicking multiple clusters before selecting this option.
- *Edit Selected Cluster* This option will allow a user to edit an existing stored cluster. Administrators can use this wizard to modify the set of hosts or services in an existing cluster. They can choose 2 ways to edit a cluster from the entry, they can re-run the cluster wizard or they can choose to use the embedded YAML editor available in the dashboard.

Figure 4.7. Select edit mode:



If we select the online editor option, it will open the selected cluster into a YAML editor window.

Figure 4.8. YAML Editor



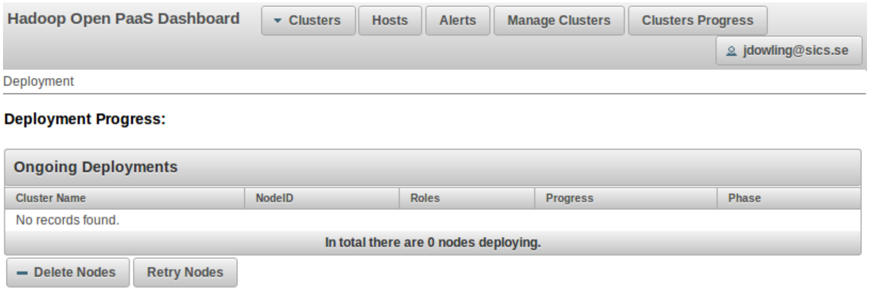
- *Load Selected Cluster* This option allows the user to load an existing cluster definition from a YAML file. This is a useful feature for reusing cluster definitions - users can share cluster definitions and load them using this option.
- *Export Cluster* It saves the information about the cluster in a YAML file, so that the cluster definition can be shared.

Cluster Deployment Progress

The Hop Dashboard monitors all the nodes deployed in its clusters. It displays the history of all the nodes going through multiple phases of the deployment cycle. A progress bar appears over each of the entries in the table showing the deployment progress of the nodes. A cluster node goes through the following phases during deployment:

- *Waiting:* An entry has been generated in the node scheduler but no node creation query has been yet been sent to the cloud provider.
- *Creation:* The scheduler has submitted the query to the cloud provider and it is waiting for the cloud provider to finish deploying the virtual instance. When an instance is successfully created an initialization script is executed for preliminary configuration of the node.
- *Install:* After the node is successfully commissioned, chef installs recipes for Hop Services to fetch the necessary binaries from the different repositories. This phase is optional in case of using pre built virtual instances which contain Hop binaries.
- *Configure:* A node in this phase means, that; the node is receiving the configuration script which will execute chef with the selected recipes for the services defined for that node.
- *Complete:* The node has successfully finished executing the configuration script with chef and it is now part of the working cluster.
- *Retrying:* The deployment system has detected a problem during a node deployment phase and it is retrying to recover from the failure. It retries the failed phase script for five times.
- *Error:* When all the retires are exhausted the node is marked erroneous. Hadoop administrator can to take further actions to recover the node, for example, SSH into the failed node and the fix the problem manually, retry the deployment process or delete the failed nodes.

Figure 4.9. Clusters Progress



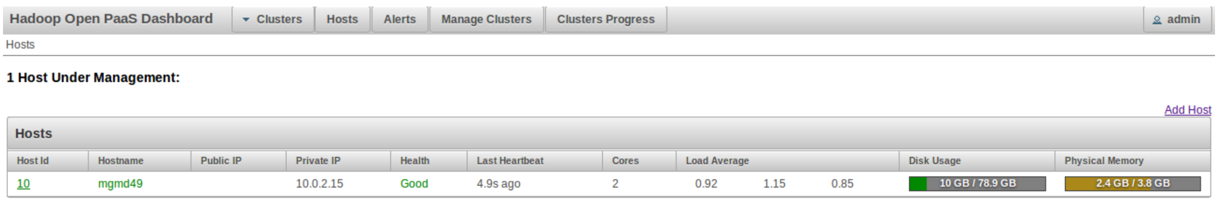
Monitoring

Hop Dashboard uses collectd to monitor hosts, alerts and Hop services in real time. A brief overview of the monitoring tools is given below.

Hosts

The host monitoring tool shows the state of all the nodes in the clusters. An administrator can track information of his/her interest such as the allocated IPs for the nodes, hostnames, host IDs, health of the nodes in the system, and when the last heartbeat was received from the nodes. It also monitors available resources on the nodes such as the number of cores that machine has, average load on the instance, disk usage and physical memory in use.

Figure 4.10. Hosts



Selecting the host ID's displays a detailed view of the host's analytics with the graphs that are available for the hosts. See the the section called “Edit Graphs” section to create or delete graphs for the dashboards monitoring components.

Figure 4.11. Hosts Details-Services

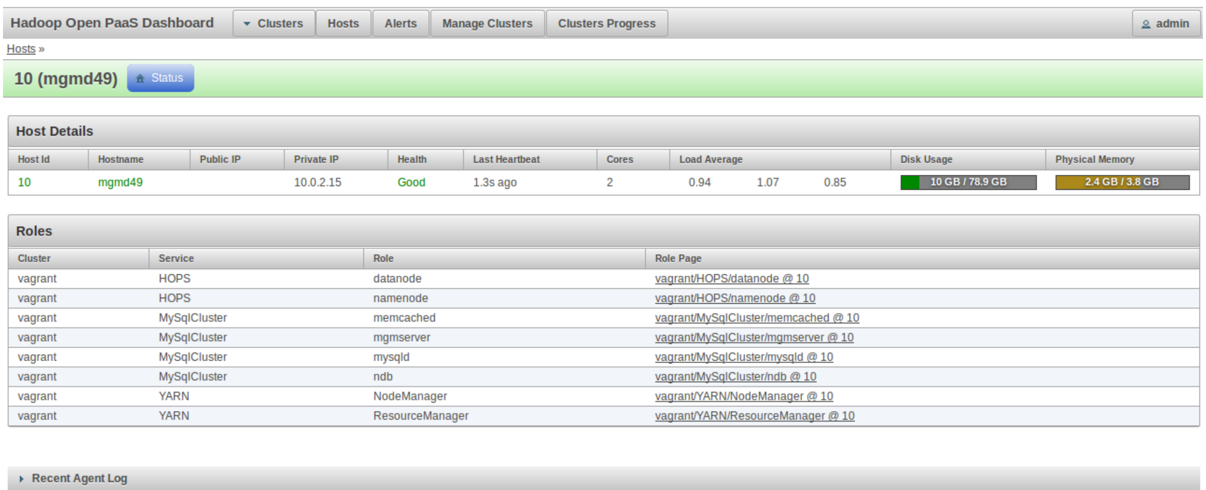
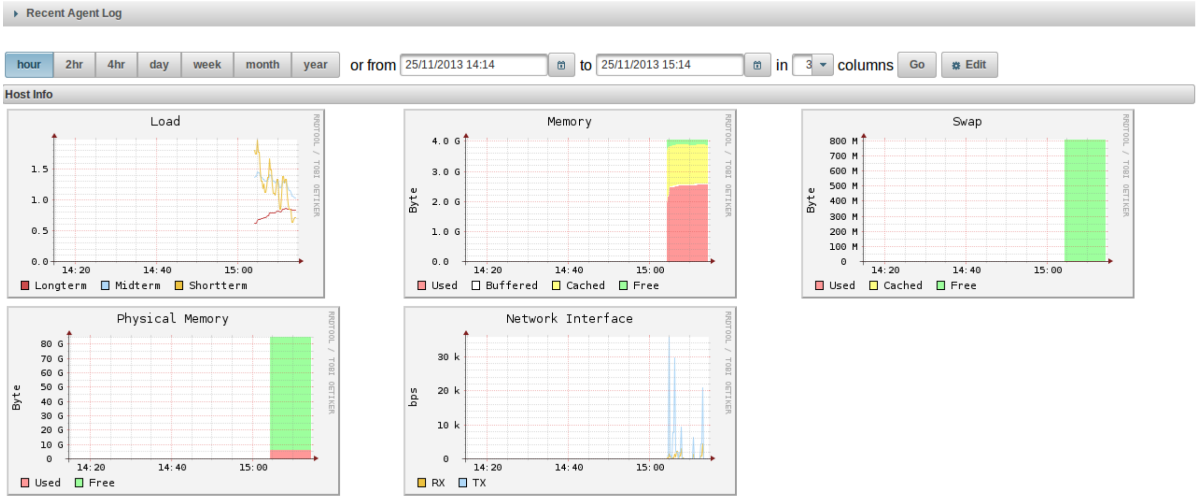


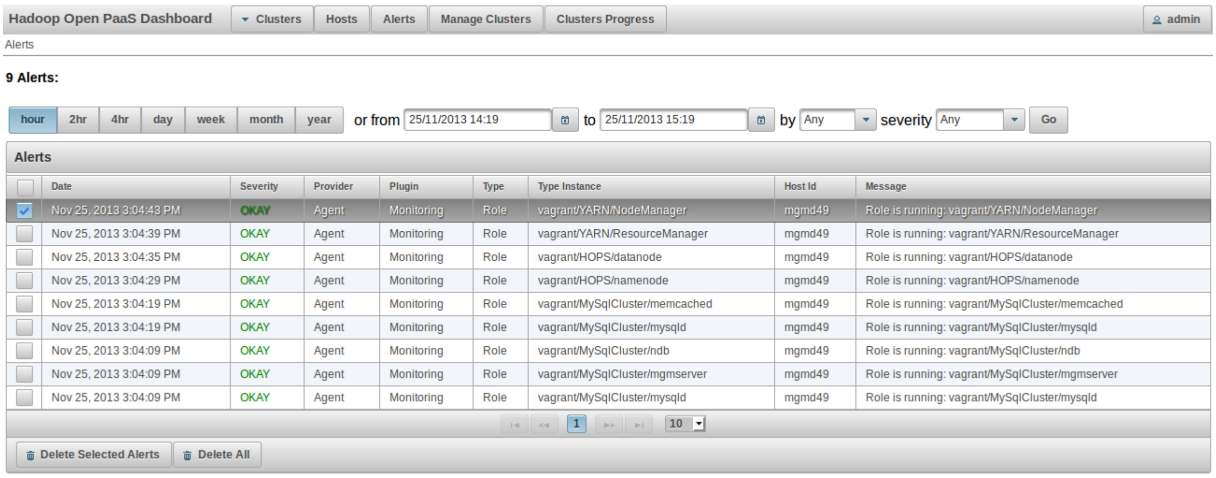
Figure 4.12. Hosts Details Graphs



Alerts

This feature allows the user to view alerts for events of interest to the administrator, such as failed hosts, full hard drives, etc. An administrator can filter alerts of certain severity and that were generated in a certain time window. The Alerts feature is based on collectd thresholds.

Figure 4.13. Alerts



Clusters

The cluster monitoring tool gives a bird eye view of the clusters monitored by the Hop Dashboard. It contains information such as the nodes that compose the cluster and the health status of the services deployed on those nodes. Also it keeps track of the resources allocated to the cluster such as total number of cores which compose the overall computing power of the cluster, the total disk capacity and the total physical memory capacity. An administrator can drill down into a cluster to find out the list of services in the cluster, their status and a list of nodes in each service. Many services can be restarted from the Dashboard.

Figure 4.14. Clusters

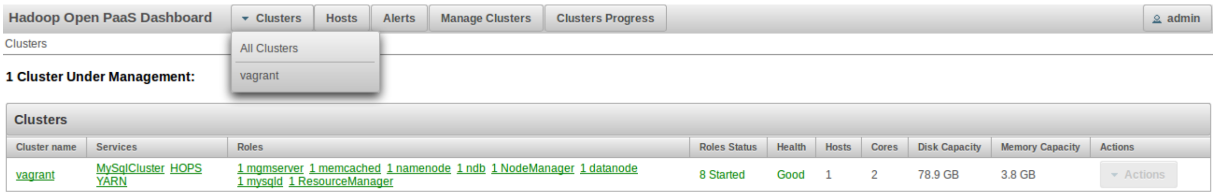
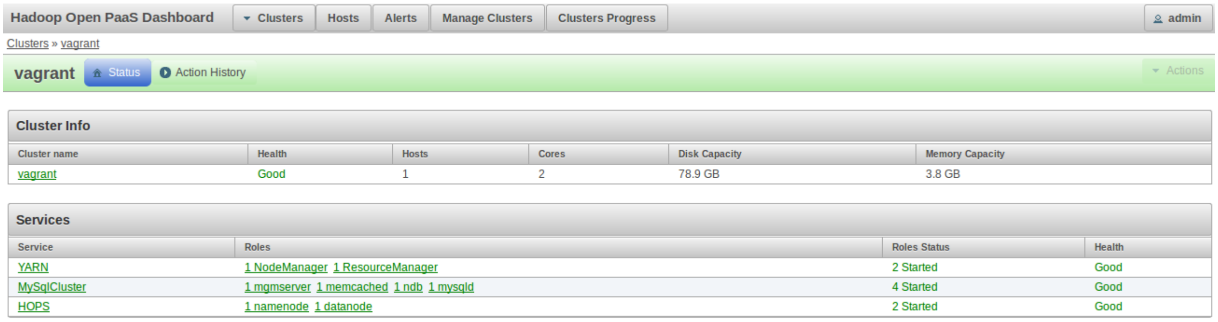


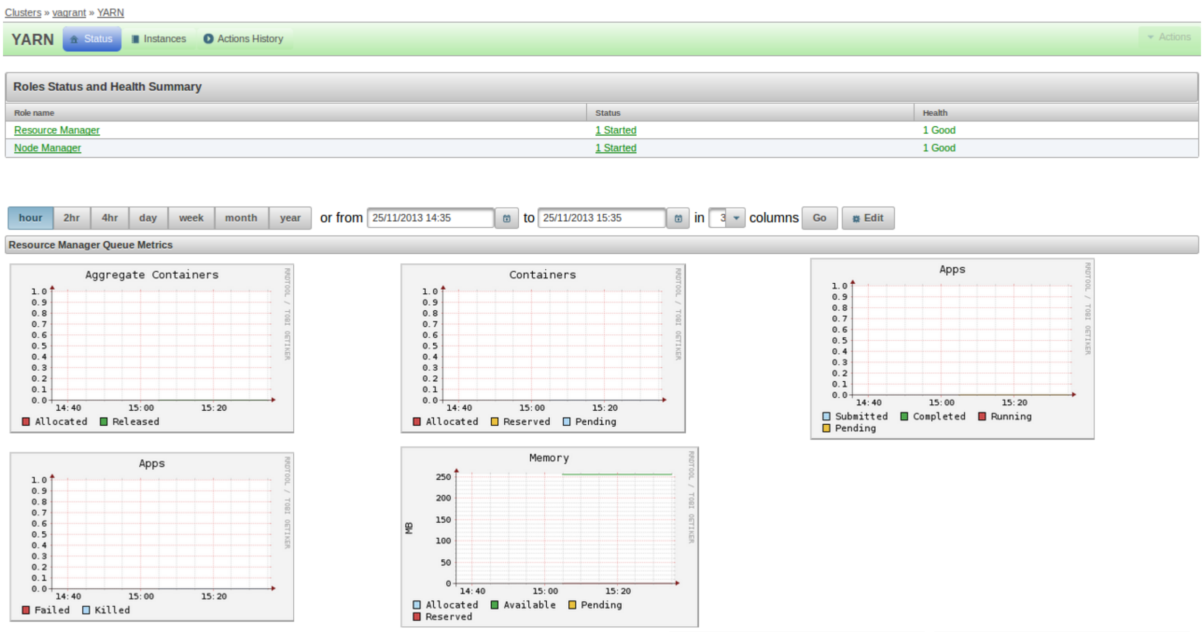
Figure 4.15. Detailed Cluster View



YARN monitoring

The Hop Dashboard displays graphs on YARN performance analytics.

Figure 4.16. YARN Metrics



You can also access the Apache websites for the resource and node managers in YARN.

Figure 4.17. Resource Manager Metrics

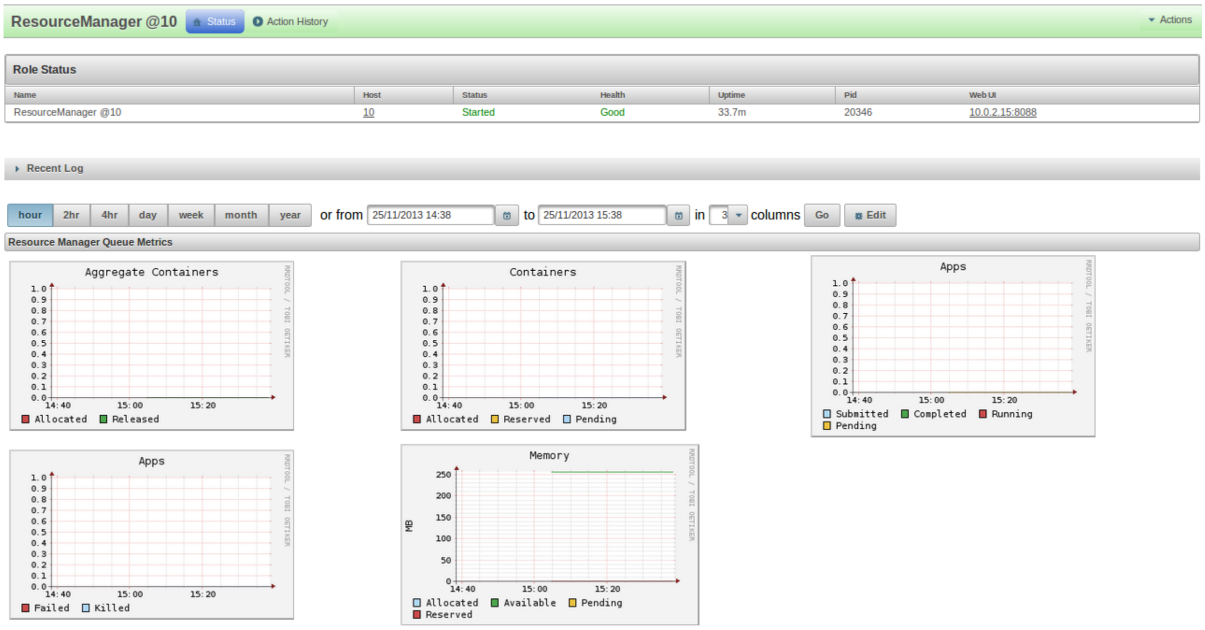
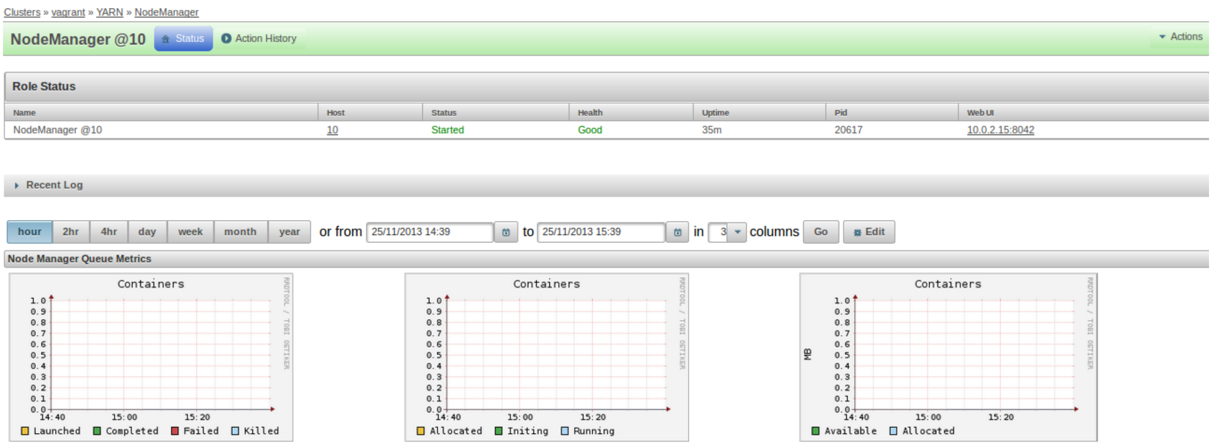


Figure 4.18. Node Manager Metrics



The Hop dashboard provides direct access to the resource and node manger web UIs provided by the Apache distribution. The resource manager web UI gives detailed information about the number of applications submitted, number of applications in progress, number of application completed, total available memory, memory consumed, dead nodes etc. The Node manager web UI displays information such as total memory allocated to the java virtual machine on the node, memory allocated to each container, node health status, hadoop version, etc.

Figure 4.19. Resource Manager UI

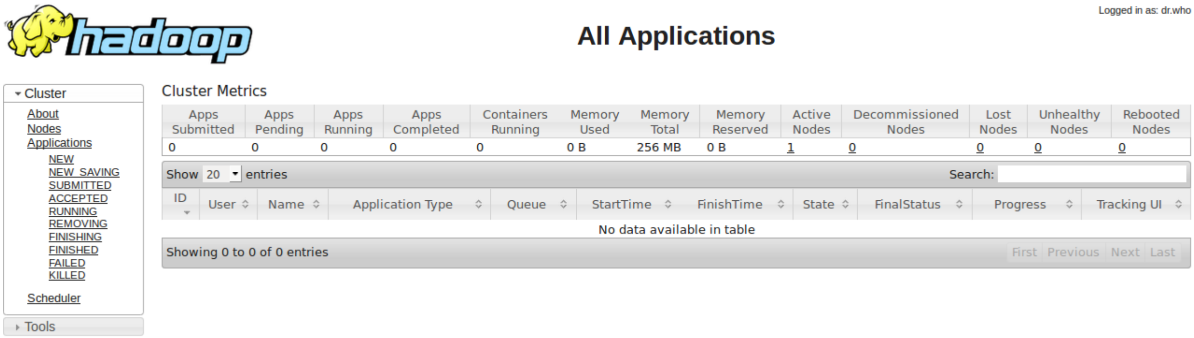
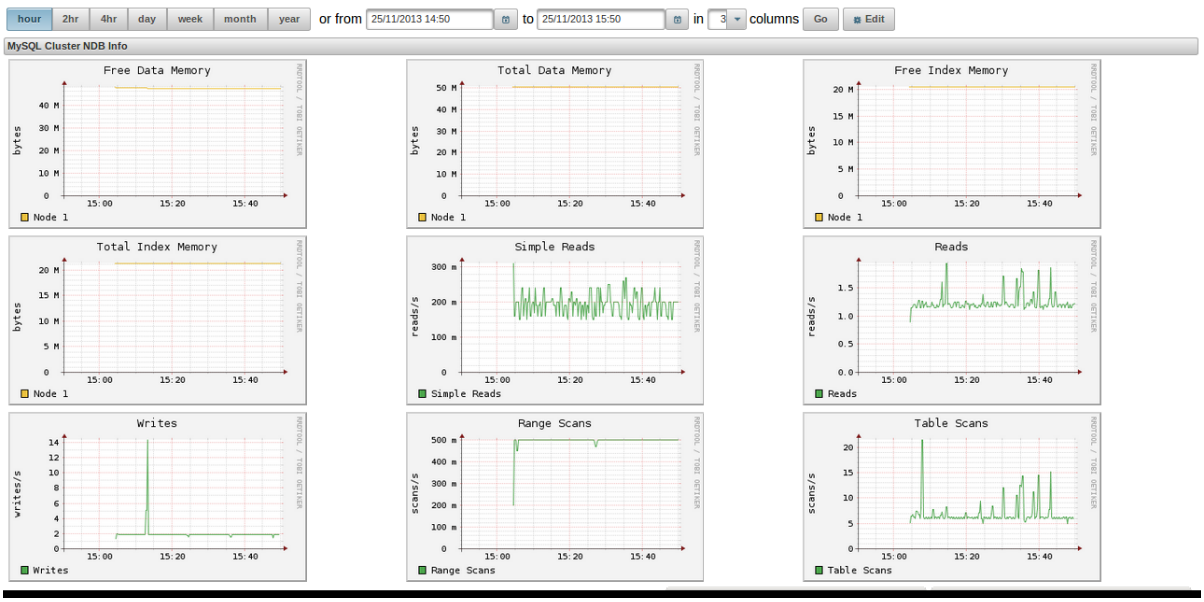


Figure 4.20. Node Manager UI

MySQL Cluster

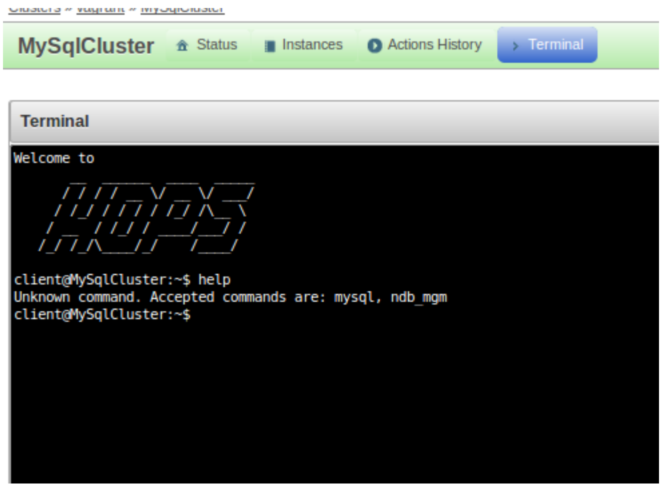
You can monitor the MySQL Cluster using the Hop Dashboard. Its keeps track of data memory, index memory, number of reads and writes per second and number of scans per second. Data nodes, management nodes, and MySQL servers can all be restarted from the Dashboard.

Figure 4.21. MySQL Cluster Monitor



The dashboard provides a commmand-line terminal to directly run SQL commands on a MySQL Server connected to NDB.

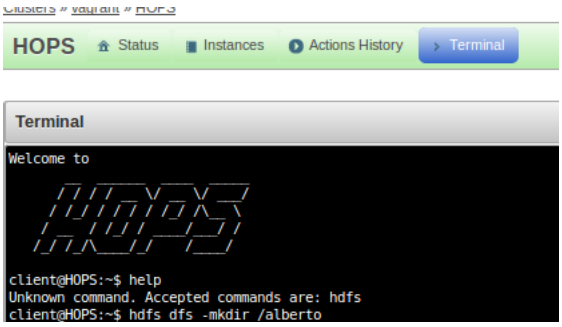
Figure 4.22. MySQL Console



Hop HDFS Console

The dashboard provides a command-line terminal for executing HDFS operations using the HDFS client. It perform operations such as ls, mkdir, cat, etc.

Figure 4.23. Hdfs Console



Chapter 5. Defining a Cluster

In this section, we describe the tools we offer in order to easily define and structure HOP clusters for their deployment through our orchestration architecture. In here, we will introduce you to our definition language to define clusters for cloud providers like Amazon EC2 and OpenStack or Baremetal clusters based on physical machines. With our cluster definition language, you will see that you will easily have a cluster deployed in a matter of minutes by making use of technologies like Chef that will be in charge of orchestrating the nodes while we provision them with Jclouds (in the case of a virtual environment) or a simple SSH client for your baremetal machines.

Cluster Definition Language

We will start first by presenting our Cluster Definition Language (CDL) with which you can defining your clusters with ease. In general, we handle the following abstractions:

- *Cluster:* A cluster is an entity that defines a whole system based on a heterogenous structure composed of multiple nodes. In most of the cases, we can classify the nodes into groups depending of the software they run. To allow further customization of your cluster, we allow interesting options like the possibility of running chef recipes globally on all the nodes and open ports that you may want to be open. In order to identify which type of cluster you are defining, it is necessary to specify the class tag of the type of cluster so our software can map with the bindings of the type of cluster you want to deploy.

Example 5.1. Defining Global Properties

```
!!se.kth.kthfsdashboard.virtualization.clusterparser.Cluster
##name of your cluster
name: test2
##enable install phase
installPhase: true
##global parameters
global:
##user defined recipes
  recipes:
  - ssh
  - chefClient
## extra ports you want to open
  authorizePorts:
  - 3306
  - 4343
  - 3321
```

Git Repositories

If you want further customization, it is possible to fork our git repository and customize our chef recipes if you want to modify some parameters of our cluster. Also you can add your own recipes if you decide to launch other services on your code. Simply add this snippet of code under global parameters.

Example 5.2. Defining Git repository

```
git:
  user: Jim Dowling
  repository: https://ghetto.sics.se/jdowling/hops-chef.git
  key: notNull
```

- *Services:* We identify multiple services, in our case related to Hop platform. You can spread this services quite easily among different nodes just indicating that information when grouping them. Also you may indicate further services to be deployed on them.

```
service:
  - datanode
  - nodemanager
  number: 2
```

- *Provider:* In the case of defining a cluster to be deployed in a virtualized environment through an Amazon EC2 infrastructure or an OpenStack environment, you can give information of the image you want to use, the type of instance to request, login credentials in case you are using custom images.

Example 5.3. Defining Cloud Providers

```
provider:
  ##name of the provider, use aws-ec2 or openstack-nova
  name: aws-ec2
  ##if EC2 use a value to one of EC2 types, in OpenStack this is an id number
  ##type of instance you want to use
  instanceType: m1.large
  ## indicate the login user of the machine with sudo access, necessary for cu
  ## or openstack image
  loginUser: ubuntu
  ## image you want in EC2 or OpenStack
  image: eu-west-1/ami-35667941
  ##region of EC2 or project name in OpenStack
  region: eu-west-1
```

We will also see that the syntax differs on whether are designing your cluster towards a virtualized environment or a physical environment. In the following sections, we will go through detailed examples for both types of clusters.

Structuring your Cluster:

Before using our tools, it is important that you have an idea of how you want to structure the services of our data platform through out the whole cluster. In our case, a fully functional cluster requires the following services deployed in different machines:

1. *MySQL Cluster:*

   - *MySQL-NDB:* Your cluster should contain at least 2 instances of NDB

- *MySQL-MGM:* Your cluster should contain at least 1 instance of a Management Server.
- *MySQL-Mysqld:* Your cluster should contain at least 1 instance of a MySQL Server.

2. *HOP*

- *Namenode:* Your cluster should contain at least 2 namenode instances of our Hadoop Solution.
- *Datanode:* Your cluster should contain at least 2 datanode instances of our Hadoop Solution.

3. *Data processing*

- *ResourceManager:* Your cluster should contain at least 1 resource manager instances of YARN.
- *NodeManager:* Your cluster should contain at least 2 node manager instances of YARN.
- *Spark:* Your cluster should contain at least one instance of Spark if you want to do data processing through Spark to submit your jobs to the system.

Multiple Services per Node

The previous section gave a very simple overview of the components that are needed for a HOP cluster to work correctly. It is possible to allocate various services in one machine or group of machines as we will see in the following sections.

Now that we have a general perspective of how a cluster looks like, the next step is to identify the environment of your choice for the cluster you want to work with. In the following sections, we will describe how you can define the structure for virtualized cloud providers like Amazon EC2 and OpenStack or in a physical Baremetal environment.

Building your cluster:

In this section, we will explain through a couple of complete examples how to define your cluster for Amazon EC2, OpenStack or Baremetal. We will show you how to write your cluster from scratch using your own YAML file or you can use the available cluster wizard in order to generate your desired cluster.

Cluster in AWS

Lets imagine that we want to define a complete HOP cluster which will contain a basic minimal setup. In this case we need 2 NDBs, 1 MGM and 1 Mysqld for the MySQL cluster, 2 namenode and 2 datanode for the Hadoop File System and in order to user Spark, a Spark instance with 1 resource manager and 2 node managers. How we could map the services using only 7 machines? A very simple configuration could be as follows:

Example 5.4. Full AWS Cluster Example

```
!!se.kth.kthfsdashboard.virtualization.clusterparser.Cluster
name: test2
provider:
  name: aws-ec2
  instanceType: m1.large
  loginUser: ubuntu
  image: eu-west-1/ami-35667941
  region: eu-west-1

##lists of groups, with the roles the nodes
##will have and open ports
nodes:
- service:
  - ndb
  number: 2

- service:
  - mgm
  number: 1

- service:
  - mysqld
  - namenode
  number: 1

- service:
  - namenode
  - resourcemanager
  number: 1

- service:
  - datanode
  - nodemanager
  - spark
  number: 2
```

With this configuration file, we will create 5 security groups which will have as a name the first service defined in the list. This will also open the ports for those security groups. It will install the defined services for each of the nodes in that specific group of nodes.

Cluster in OpenStack

Taking the previous case for Amazon EC2, we can easily write the same cluster description using the same cluster definition file. In this case, the only section we need to change is related to the provider we want to use which in this case is OpenStack. The file will look as follows:

Example 5.5. Full OpenStack Example

```
!!se.kth.kthfsdashboard.virtualization.clusterparser.Cluster
name: nova
provider:
  name: openstack-nova
  instanceType: 7
  loginUser: ubuntu
  image: 0190f9c4-d64e-4412-ab88-4f9fd1d7c2e3
  region: RegionSICS

##lists of groups, with the roles the nodes
##will have and open ports
nodes:
  - service:
    - ndb
    number: 2

  - service:
    - mgm
    number: 1

  - service:
    - mysqld
    - namenode
    number: 1

  - service:
    - namenode
    - resourcemanager
    number: 1

  - service:
    - datanode
    - nodemanager
    - spark
    number: 2
```

With this configuration file, it is possible to deploy the same cluster we defined in Amazon EC2 without any major changes. You only need to change the provider specifications to match the details of your OpenStack Infrastructure.

Cluster on Baremetal Machines

How would we describe the same cluster for Amazon EC2 in a cluster of physical machines? In this case it is much simpler but you need to watch out for minor details like, for example; the class tag needs to be different for this type of clusters as we will see. Also in this case, you need to provide the IP addresses of the machines to connect to. An example is as follows:

Example 5.6. Full Baremetal Example

```
!!se.kth.kthfsdashboard.virtualization.clusterparser.Baremetal
name: baremetal
loginUser: ubuntu
totalHosts: 7
nodes:
  - service: ndb
    number: 2
    hosts:
    - 10.20.0.8
    - 10.20.0.11

  - service: mgm
    number: 1
    hosts:
    - 10.20.0.6

  - service:
    - mysqld
    - namenode
    number: 1
    hosts:
    - 10.20.0.7

  - service:
    - namenode
    - resourcemanager
    number: 1
    hosts:
    - 10.20.0.12
    - 10.20.0.14

  - service:
    - datanode
    - nodemanager
    - spark
    number: 2
    hosts:
    - 10.20.0.16
    - 10.20.0.17
```

With this configuration file, it is possible to deploy the same cluster we defined in Amazon EC2 without any major changes. You only need to change the provider specifications to match the details of your OpenStack Infrastructure.

Cluster Generator on Dashboard

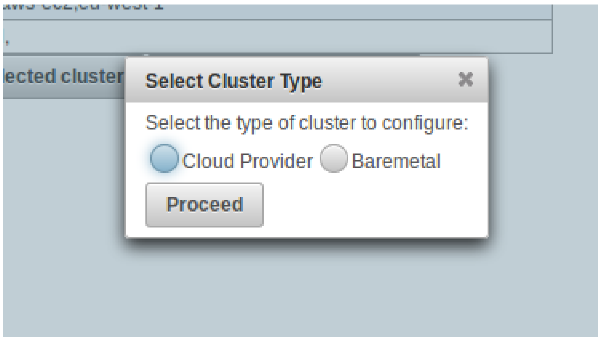
Apart of offering a mechanism were users can upload their clusters written in YAML to the system and later on deploy them, we also have a cluster wizard which allows the user to define a cluster step by step quite easily. To make use of this feature, follow these steps:

1. Go to the manage cluster section from the main bar in the dashboard. Select the create cluster option. Main Menu Bar → Manage Cluster → Create cluster

2. A dialog appears allowing you to select which type of cluster you want to use:

- *Virtualized:* Choose this option if you want to deploy a cluster in Amazon EC2 or OpenStack.
- *Baremetal:* Choose this option if you want to deploy a cluster in physical machines.

Figure 5.1. Select Cluster Type:



3. Selecting an option, will bring you to the cluster generator wizard. Here you can select the same options like if you where writing your own file from scratch. You will go through different phases.

   Cluster Wizard → Common → Provider (not for Baremetal) → Groups → Confirmation

   - *Common Section:* In this section, a form appears were you can select the following options:

     a. *Name:*Name of the cluster

     b. *Provider:*Select the type provider between Amazon EC2 or OpenStack, this option is available if we create a virtualized cluster.

     c. *Git parameters:* Git repository section where you can specify as an option your own git repository based on our code. This way you can customize our recipes or even add your own.

     d. *Global Recipes:* You can specify chef recipes that you want to execute in all the nodes

     e. *Global Ports:* Additional Ports to open for your cluster, this option is only available for virtualized clusters.

Figure 5.2. Common Cluster Options:

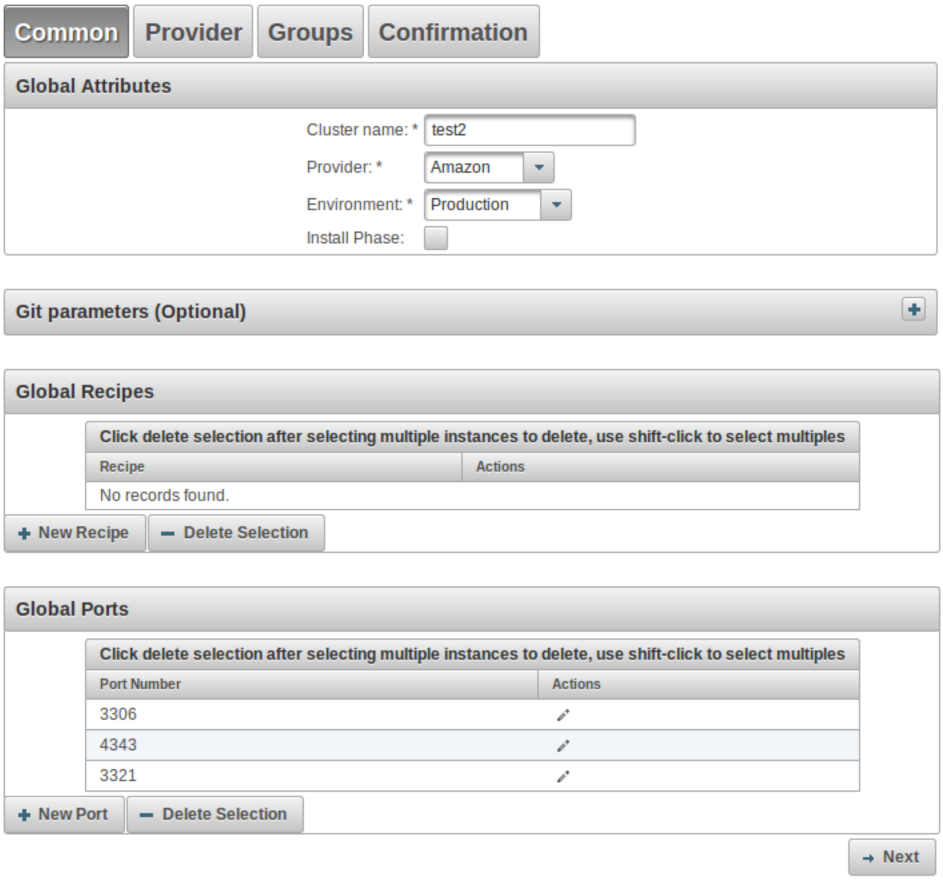
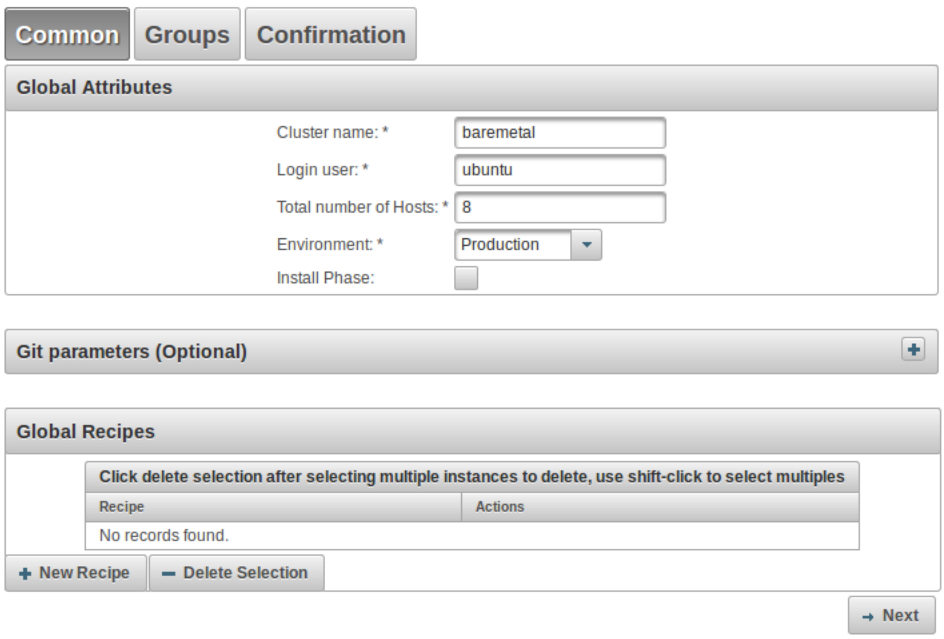


Figure 5.3. Bare Metal Common Cluster Options:



- *Provider Section:* This form enables you to define the parameters for OpenStack or Amazon EC2. Some values appear by default in the case of Amazon EC2 of defining a cluster to be used with this cloud provider.

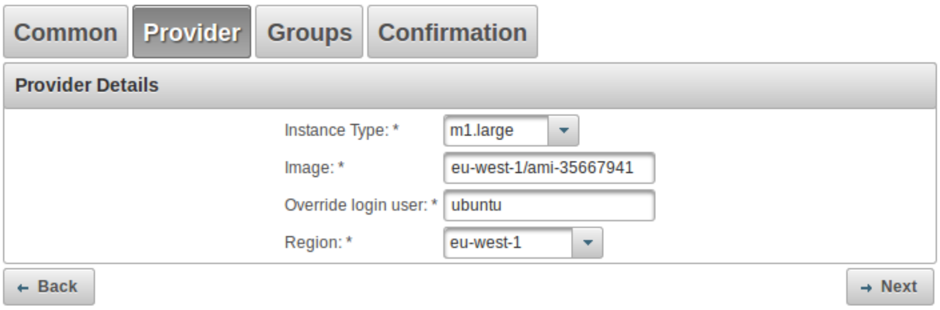
a. *Instance Type:* The type of instance you want to use in Amazon EC2 or in OpenStack. Note that in OpenStack we use the id number of the type of instance, not the name.

b. *Image:* The name of the image we want to use the in Amazon EC2 or in OpenStack

c. *Login user:* Here you include the user name with sudo access to access the instances in Amazon EC2 or OpenStack. Note that this value is necessary if you use a custom AMI in Amazon EC2 or using you use OpenStack.

d. *Region:* Here you include the region you want to deploy in Amazon EC2 or the project to use in your OpenStack infrastructure.

Figure 5.4. Cluster Provider Options:



- *Group Section:* In this section you can specify the group of nodes for you cluster with the their services and ip addresses (if you are deploying a baremetal cluster)

a. *Main Service:* The main service you want to deploy in this group of nodes

b. *Bittorrent Support:* If you want to enable bittorrent sync of binaries from the dashboard.

c. *Number of nodes:* Number of nodes that will contain the same set of services.

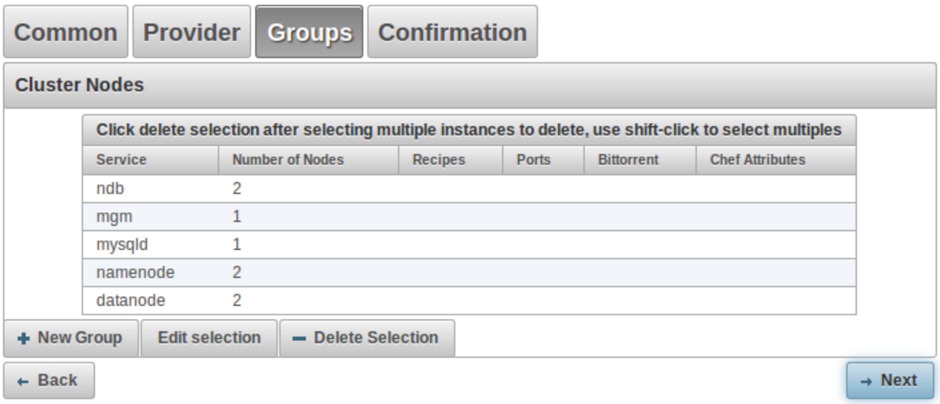
d. *Extra Services:* Other services you may want to run which can be also your own services.

e. *Chef Attributes:* In this section, you would include a chef json which will contain the attributes you may want to override from your recipes.

f. *Ports:* Extra ports that you may want to enable in that group, in this case this only affect virtualized clusters.

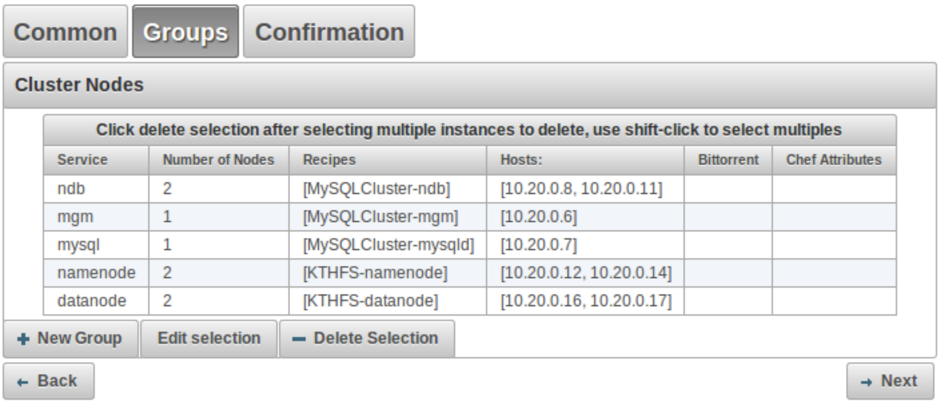
g. *Hosts:* List of hosts IP addresses for the nodes that will be part of this group of nodes. In this case this option is only available for Baremetal clusters.

Figure 5.5. Cluster Group:



Service	Number of Nodes	Recipes	Ports	Bittorrent	Chef Attributes
ndb	2				
mgm	1				
mysqld	1				
namenode	2				
datanode	2				

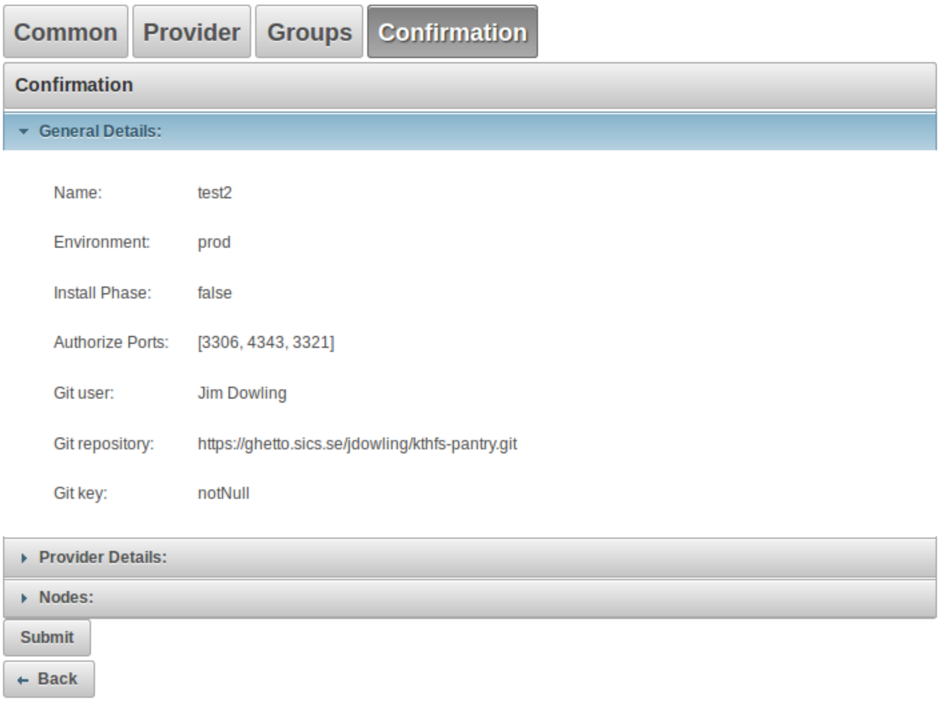
Figure 5.6. Bare Metal Groups:



Service	Number of Nodes	Recipes	Hosts:	Bittorrent	Chef Attributes
ndb	2	[MySQLCluster-ndb]	[10.20.0.8, 10.20.0.11]		
mgm	1	[MySQLCluster-mgm]	[10.20.0.6]		
mysql	1	[MySQLCluster-mysqld]	[10.20.0.7]		
namenode	2	[KTHFS-namenode]	[10.20.0.12, 10.20.0.14]		
datanode	2	[KTHFS-datanode]	[10.20.0.16, 10.20.0.17]		

- *Confirmation Section:* In this section you will see a summary of the details of you cluster file. When you press the submit button, your cluster file will be stored in the dashboard and it will proceed to the cluster launcher.

Figure 5.7. Confirmation:



Wrap up

To summarize this section, in here we have seen the main building blocks that we need to define a cluster using our cluster domain specific language. We also explained how you can define your clusters by writing your own cluster file through multiple examples and also showed an alternative way of defining cluster through the cluster generator wizard which is accessible from the dashboard.

Chapter 6. Launching a Cluster

Installation on AWS

In this section, we will explain furthers steps that are required to deploy a whole functional cluster running our data platform through the dashboard. Also we refer to recommendations and aspects you should consider before deploying a cluster.

Pre-requisites:

Before starting, make sure that you have access to a functional and running Dashboard in a virtual machine in an accessible Amazon EC2 region. If you have not done so, please refer back to the Chapter 3, *Hop Web Portal*.

Requirements:

In order to install and deploy a cluster, you need to define before the structure of the cluster which includes specifying the number of machines to create in EC2 with the specific instance type with the specific software. This can be done using a cluster definition file that can be done from scratch or using the embedded wizard available on the dashboard. Further information about describing a cluster can be found on the cluster configuration section. Before continuing make sure that you have the following.

- Cluster definition for EC2 (see related section) in a file or loaded from the dashboard database.
- Amazon EC2 credentials to deploy the cluster in Amazon, configured in the dashboard. In order to do it, select the option setup credentials found in you user icon to specify the EC2 credentials to be used by the dashboard.

Additional dashboard credentials

It is possible to include other options when deploying an EC2 cluster, for example; for maintanance purposes you might want to authorize extra public keys to the virtual machines. This is possible to set in the credential section of the dashboard.

Launching the cluster

Once we have the dashboard configured with the Amazon EC2 credentials, you can proceed to launch a cluster:

1. Select the manage cluster option available in the dashboard.

   Main Menu Bar → Manage Cluster → Load File

2. In this new view, you can manage available clusters that you may have defined previously. You can select a previous cluster, create a new one with the only wizard or load a cluster from a cluster definition file. For further information on managing cluster files, see Chapter 5, *Defining a Cluster*. To continue, select a cluster from the table or load a cluster from a file.

3. The file is loaded and the launcher view should appear. Here you can view the contents of the cluster to be deployed before launch.

4. Pressing the start cluster will start the deployment process. A status bar will appear giving information of the current status of the deployment. Also a progress table on the background will be generated with information of the configuration state of the nodes. The process is long and it depends on the number of nodes you deploy. On average, for 8 nodes it takes around 35 minutes.

Error Nodes

It is possible that some nodes will have issues during the deployment of our software (package configuration problem, erratic behaviour) which in this case our system will detect and will retry to relaunch the software on that specific machine automatically. The maximum number of retries specified for each node is 5, after that; the node will be tagged as an error node and it is possible to do a manual retry after the whole process has finished.

5. When the process completes, it will take you back to the progress view where you can see details of the cluster deployment. If nodes failed, you can select those nodes and try to recover them using the retry nodes option.

Retrying Nodes

Retrying node is an option that helps bringing back nodes that had minor issues when installing packages, were to slow to finish the configuration phase or the default number of retries we use were not enough. It will not bring back nodes which had a critical configuration failure, which in this case it will be necessary to log in directly through SSH to the specific machine in order to fix it.

Congratulations, if everything went okay; you have succesfully deployed a complete cluster ready to use!

Installation on OpenStack

In this section, we will explain furthers steps that are required to deploy a whole functional cluster running our data platform through the dashboard. Also we refer to recommendations and aspects you should consider before deploying a cluster in OpenStack.

OpenStack Deployment

Note please that this option is currently in development phase and from our tests we managed to deploy functional testing clusters. Still due to issues we encountered during our tests in our personal OpenStack, we cannot guarantee the same level of performance as deploying for example a cluster in EC2. This is due to the fact that our deployment system depends greatly on how effectively OpenStack behaves with your hardware and so unexpected behaviour might take place. If you have a very good OpenStack infrastructure, we invite you test it.

Pre-requisites:

Before starting, make sure that you have access to a functional and running Dashboard in a virtual machine accesible from you OpenStack Infrastructure. If you have not done so, please refer back to the section Chapter 3, *Hop Web Portal*.

Requirements:

In order to install and deploy a cluster, you need to define before the structure of the cluster which includes specifying the number of machines to create in OpenStack with the specific instance type with the specific software. This can be done using a cluster definition file that can be done from scratch or using the embedded wizard available on the dashboard. Further information about describing a cluster can be found on Chapter 5, *Defining a Cluster*. Before continuing make sure that you have the following.

- Cluster definition for OpenStack (see related section) in a file or loaded from the dashboard database.

- OpenStack credentials to deploy the cluster on your OpenStack infrastructure, configured in the dashboard. In order to do it, select the option setup credentials found in you user icon to specify the OpenStack credentials to be used by the dashboard.

Additional dashboard credentials

It is possible to include other options when deploying a OpenStack cluster, for example; for maintanance purposes you might want to authorize extra public keys to the virtual machines. This is possible to set in the credential section of the dashboard

Launching the cluster

Once we have the dashboard configured with the OpenStack credentials, you can proceed to launch a cluster:

1. Select the manage cluster option available in the dashboard.

   Main Menu Bar → Manage Cluster → Load File

2. In this new view, you can manage available clusters that you may have defined previously. You can select a previous cluster, create a new one with the only wizard or load a cluster from a cluster definition file. For further information on managing cluster files, see the cluster configuration section. To continue, select a cluster from the table or load a cluster from a file.

3. The file is loaded and the launcher view should appear. Here you can view the contents of the cluster to be deployed before launch.

4. Pressing the start cluster will start the deployment process. A status bar will appear giving information of the current process. Also a progress table on the background will be generated with information of the configuration state of the nodes. The process is long and it depends on the number of nodes you deploy. On average, for 8 nodes it takes around 35 minutes.

   ### Error Nodes

   It is possible that some nodes will have issues during the deployment of our software (package configuration problem, erratic behaviour) which in this case our system will detect and will retry to relaunch the software on that specific machine automatically. The maximum number of retries specified for each node is 5, after that; the node will be tagged as an error node and it is possible to do a manual retry after the whole process has finished.

5. When the process completes, it will take you back to the progress view where you can see details of the cluster deployment. If nodes failed, you can select those nodes and try to recover them using the retry nodes option.

   ### Retrying Nodes

   Retrying node is an option that helps bringing back nodes that had minor issues when installing packages, were to slow to finish the configuration phase or the default number of retries we use were not enough. It will not bring back nodes which had a critical configuration failure, which in this case it will be necessary to log in directly through SSH to the specific machine in order to fix it.

Congratulations, if everything went okay; you have succesfully deployed a complete cluster ready to use!

Installation on Baremetal Machines

In this section, we explain the steps that are required through the dashboard to deploy our data platform on a cluster of hosts running the linux operating system.

Pre-requisites:

Before starting, make sure that you have access to a functional and running Dashboard in a host which you can access via a browser. If you have not done so, please refer back to Chapter 3, *Hop Web Portal*.

Requirements:

In order to install and deploy a cluster, you first need to specify the set of ip addresses for the hosts and the specific software. This can be done using a cluster definition file that can be done from scratch or using the embedded wizard available on the dashboard. Further information about describing a cluster can be found on Chapter 5, *Defining a Cluster*. Before continuing make sure that you have the following.

- Cluster definition for a baremetal cluter (see related section) in a file or loaded from the dashboard database.
- Credentials to connect to your physical machine, this means a user name with sudo access and the private key to SSH the machines.

Launching the cluster

Once we have the dashboard configured with the physical machines credentials, you can proceed to launch a cluster:

1. Select the manage cluster option available in the dashboard.

   Main Menu Bar → Manage Cluster → Load File

2. In this new view, you can manage available clusters that you may have defined previously. You can select a previous cluster, create a new one with the only wizard or load a cluster from a cluster definition file. For further information on managing cluster files, see the cluster configuration section. To continue, select a cluster from the table or load a cluster from a file.

3. The file is loaded and the launcher view should appear. Here you can view the contents of the cluster to be deployed before launch.

4. Pressing the start cluster will start the deployment process. A status bar will appear giving information of the current process. Also a progress table on the background will be generated with information of the configuration state of the nodes. The process is long and it depends on the number of nodes you deploy. On average, for 8 nodes it takes around 35 minutes.

   ### Error Nodes

   It is possible that some nodes will have issues during the deployment of our software (package configuration problem, erratic behaviour) which in this case our system will detect and will retry to relaunch the software on that specific machine automatically. The maximum number of retries specified for each node is 5, after that; the node will be tagged as an error node and it is possible to do a manual retry after the whole process has finished.

5. When the process completes, it will take you back to the progress view where you can see details of the cluster deployment. If nodes failed, you can select those nodes and try to recover them using the retry nodes option.

   ### Retrying Nodes

   Retrying node is an option that helps bringing back nodes that had minor issues when installing packages, were to slow to finish the configuration phase or the default number of retries we use were not enough. It will not bring back nodes which had a critical configuration failure, which in this case it will be necessary to log in directly through SSH to the specific machine in order to fix it.

Congratulations, if everything went okay you have succesfully deployed a cluster that is ready to use!

Chapter 7. Configuring HDFS

We introduce a few new configuration parameters to HDFS, due to our support for multiple NameNodes and use of MySQL Cluster for metadata storage. These parameters are specified in *hdfs-site.xml*. The configuration parameters listed below are additional to the configuration parameters for vanilla HDFS [http://hadoop.apache.org/docs/current/hadoop-project-dist/hadoop-hdfs/hdfs-default.xml].

HDFS Configuration Parameters not used

We have replaced HDFS 2.x's Primary-Secondary Replication model with shared atomic transactional memory. This means that we no longer use the parameters in HDFS that are based on the (eventually consistent) replication of *edit log entries* from the Primary NameNode to the Secondary NameNode using a set of quorum-based replication servers. Here are the parameters that are not used in the HOP version of HDFS 2.x:

- *dfs.namenode.secondary.*:* None of the secondary NameNode attributes are used.
- *dfs.namenode.checkpoint.*:* None of the checkpoint attributes are used.
- *dfs.image.*:* None of the FSImage attributes are used.
- *dfs.journalnode.*:* None of the hadoop's journaling attributes are used.
- *dfs.ha.*:* None of the hadoop high availability attributes are used.
- *dfs.namenode.num.extra.edits.*:* None of the edit logs attributes are used.
- *dfs.namenode.name.dir.** FSImage is not supported anymore.
- *dfs.namenode.edits.** None of the edit log attributes are used.
- *dfs.namenode.shared.edits.** None of the edit log attributes are used.

Additional HDFS Configuration Parameters

- *dfs.storage.type:* In HOP all the NameNodes in the system are stateless. All the file system metadata is stored in a relational database. We have chosen MySQL NDB Cluster for its high performance and availability for the storage of the metadata. However the metadata can be stored in any relational database. Default value is this parameter is ‘clusterj’. By default HOPS uses ClusterJ libraries to connect to MySQL NDB Cluster. Later we will provide support of other DBMSs.
- *dfs.dbconnector.string:* Host name of management server of MySQL NDB Cluster.
- *dfs.dbconnector.database* Name of the database that contains the metadata tables.
- *dfs.dbconnector.num-session-factories* This is the number of connections that are created in the ClusterJ connection pool. If it is set to 1 then all the sessions share the same connection; all requests for a SessionFactory with the same connect string and database will share a single SessionFactory. A setting of 0 disables pooling; each request for a SessionFactory will receive its own unique SessionFactory. We set the default value of this parameter to 3.
- *dfs.storage.mysql.user:* A valid user name to access MySQL Server. For higher performance we use MySQL Server to perform a aggregate queries on the file system metadata.
- *dfs.storage.mysql.user.password:* MySQL user password
- *dfs.storage.mysql.port:* MySQL Server port. If not specified then default value of 3306 is chosen.

- *dfs.quota.enabled:* Using this parameter quota can be en/disabled. By default quota is enabled.
- *dfs.namenodes.rpc.address:* HOP support multiple active NameNodes. A client can send a RPC request to any of the active NameNodes. This parameter specifies a list of active NameNodes in the system. The list has following format [ip:port, ip:port, …]. It is not necessary that this list contain all the active NameNodes in the system. Single valid reference to an active NameNode is sufficient. At the time of startup the client will obtain the updated list of all the NameNodes in the system from the given NameNode. If this list is empty then the client will connect to 'fs.default.name'.
- *dfs.namenode.selector-policy:* For a RPC call client will choose an active NameNode based on the following policies.

  1. ROUND_ROBIN

  2. RANDOM
  By default NameNode selection policy is set of ROUND_ROBIN
- *dfs.leader.check.interval:* One of the active NameNodes is chosen as a leader to perform housekeeping operations. All NameNodes periodically send a HeartBeat and check for changes in the membership of the NameNodes. By default the HeartBeat is sent after every second. Increasing the time interval would lead to slow failure detection.
- *dfs.leader.missed.hb:* This property specifies when a NameNode is declared dead. By default a NameNode is declared dead if it misses a HeatBeat. Higher values of this property would lead to slow failure detection.
- *dfs.block.pool.id:* Due to shared state among the NameNodes, HOP only support one block pool. Set this property to set a custom value for block pool. Default block pood id is HOP_BLOCK_POOL_123.
- *dfs.name.space.id:* Due to shared state among NameNodes, HOP only support one name space. Set this property to set a custom value for name space. Default name space id is 911 :)
- *dfs.clinet.max.retires.on.failure:* The client will retry the RPC call if the RPC fails due to the failure of the NameNode. This property specifies how many times the client would retry the RPC before throwing an exception. This property is directly related to number of expected simultaneous failures of NameNodes. Set this value to ‘1’ in case of low failure rates such as one dead NameNode at any given time. It is recommended that this property must be set to value >= 1.
- *dsf.client.max.random.wait.on.retry:* A RPC can fail because of many factors such as NameNode failure, network congestion etc. Changes in the membership of NameNodes can lead to contention on the remaining NameNodes. In order to avoid contention on the remaining NameNodes in the system the client would randomly wait between [0,MAX_VALUE] ms before retrying the RPC. This property specifies MAX_VALUE; by default it is set to 1000 ms.
- *dsf.client.refresh.namenode.list:* All clients periodically refresh their view of active NameNodes in the system. By default after every minute the client checks for changes in the membership of the NameNodes. Higher values can be chosen for scenarios where the membership does not change frequently.