CloudKon

Distributed Task Scheduling in Cloud

Rajagopal Parthasarathi

Department of Computer Science

Illinois Institute of Technology

Chicago, IL USA

rparthas@hawk.iit.edu

Kamal Nath N.G. Shenoi

Department of Computer Science

Illinois Institute of Technology

Chicago, IL USA

kninthas@hawk.iit.edu

**ABSTRACT**

Predictions are that by the end of this decade, we will have exascale system with millions of nodes and billions of threads of execution [1]. Task Scheduling and execution of tasks over these large scale, distributed systems plays an important role in achieving good performance and high system utilization. Many Task computing paradigm[2] aims to bridge the gap between High Performance Computing and High Throughput Computing . Tasks may be small or large, uniprocessor or multiprocessor, compute-intensive or [data-intensive](http://en.wikipedia.org/wiki/Data-intensive_computing) but MTC tasks include loosely coupled communication intensive tasks. Today’s job schedulers have centralized Master/Slaves architecture (e.g. Slurm, Condor, PBS,SGE), where a centralized server is in charge of the resource provisioning and job execution. This architecture has worked well in modest scales and coarse granular workloads, but it has poor scalability at the extreme scales of petascale systems with fine granular MTC workloads. The goal of this project is to provide a efficient light weight and scalable distributed execution framework built on built on open source stack[HazelCast,Cassandra] to address MTC workloads deployed over Amazon Ec2 instance in cloud environment

**Index Terms**— Cloud Computing, Many Task Computing, Distributed Scheduling, Task Execution

# INTRODUCTION

The goal of an execution fabric is to effectively utilize the execution system aiming towards high throughput and provide efficient results for executed tasks. Today’s workload involves many fine granular workloads with execution times in seconds. Centralized schedulers are optimized towards high computational massive tasks where the complex decision policy and architecture of the schedulers play a major role. However, they tend to add considerable overhead while scheduling these lots of small tasks. Moreover, the centralized architecture tends to be a bottleneck in scheduling and execution. The solution to this problem is to have a decentralized and simple architecture. A decentralized architecture avoids the single point of failure, while a simple architecture reduces the considerable overhead involved in decision making for scheduling.

An execution fabric requires lot of computing resources to address the ever-growing workload of today’s world. Clouds seem to be a viable solution to this problem. Our solution is to build a loosely coupled compact and distributed execution fabric over public cloud (Amazon Ec2 instance) with distributed building blocks such as Cassandra [3] and HazelCast [4]. The motivation behind using open source stack in favor of extensive use of AWS [Amazon Web Services] is to decouple the fabric from AWS and provide easier transition to private cloud environment.

Recent studies suggest that clouds were not suitable candidates for scientific HPC computing [5][6][7][8].The problems listed were largely because of following the same approach involved in traditional clusters and grids. Clouds differ a lot from HPC applications as they are based on virtualization and shared resources.Our work involves running applications optimized for cloud environment. Traditional workloads can also be run on our execution fabric but with suitable decomposition of the workload at the client side.

In this project we implement a scalable distributed task execution framework. We have made extensive use of HazelCast , a highly scalable data distribution platform which acts as the reliable storage for the executable tasks . Our next building block is Cassandra, a distributed NoSQL store offering scalability and high availability for monitoring of the entire system. We also leverage the Amazon Elastic Compute Cloud (EC2) to manage virtual resources.

Today’s data analytics are moving towards shorter jobs with higher throughput and shorter latency. More applications are moving towards running higher number of jobs in order to improve the application throughput and performance. The focus is shifting towards Many Task Computing paradigm. Many task computing includes loosely coupled applications that are generally communication intensive.

We propose CloudKon as a job management system that achieves good load balancing and high system utilization. The heart of the CloudKon is the distributed queuing service. We have used HazelCast to facilitate this purpose. HazelCast performs the role of a highly available and reliable distributed pool of tasks to perfection. Worker Nodes are not adminstered by a centralized dispatcher or scheduler. Our work proposes an efficient pull architecture i.e worker nodes pull the tasks from the pool if they are idle. The system is loosely coupled and each component can be scaled based on the needs.

The driving factors behind our implementation of CloudKon are:

1. Design and architect a light-weight task execution framework for MTC workloads
2. Design a simple execution framework with a no frills user interface
3. Design a robust framework which can easily switch between public and private cloud environments
4. Design an extremely scalable execution framework
5. Design a loosely coupled framework to support future enhancements
6. Evaluate CloudKon with other state-of-the-art task execution systems
7. Deliver excellent throughput with <5% codebase of the job management systems

The remaining sections of this paper are as follows. Section 2 provides more background about the systems and the concepts that are related to this project and are necessary to know about. Section 3 discusses about the design and implementation details of CloudKon. Section 4 evaluates the performance of the CloudKon in different aspects using differentmetrics. Section 5 studies the related work in the area of task execution systems. Finally section 6 discusses about the limitations of the current work, and covers the future directions of this work.

# Background

## Amazon Elastic Compute Cloud (EC2)

Cloud computing services are broadly categorized into three layers

* Infrastructure-as-a-Service (IaaS)
* Platform-as-a-Service (PaaS)
* Software-as-a-Service (SaaS)

The focus of this project is on IaaS since the scientific computing community mostly focuses on IaaS because of the need for compatibility with legacy applications and systems. Amazon Elastic Compute Cloud (Amazon EC2) [6] is a web service that provides resizable compute capacity in the cloud. It is designed to make web-scale computing easier for developers. Amazon EC2 presents a true cloud hosting service for users. It allows users to use web service interfaces to launch instances with a variety of operating systems, load them with custom application environment, manage network’s access permissions, and run the image using as many systems as desired.

Amazon uses XEN hypervisor [7] as a middleware to run multiple Virtual Machines on their physical infrastructure. EC2 provides a web service that allows anyone to run their own applications on Amazon’s computing infrastructure, by letting customers “rent” computing resources by the hour. Clients are given access to an “unlimited” source of compute capacity, which is delivered through what is known as EC2 instance. An instance is a running virtual machine on Amazon’s cloud platform. Each of these instances are deployed with an Amazon Machine Image (AMI), which is just a pre-configured operating system and some bundled application software.

There exist several types of instances, each of them with different compute capacities, memory size, I/O performance and storage. Users launch one or more instances by specifying the instance type. Then the instances will be deployed on the server and user can connect to them via SSH using their public IP address. Amazon guarantees the availability rate of 99.95% in its Service Level Agreement. That means the instances are guaranteed to be available 99.95% of the time.

Considering the ways we can have access to these instances, we can categorize them in three different types:

* **Reserved instances:** Amazon allows us to pay upfront per each instance that we want to use during a given period, and in exchange, they give us a lower hourly cost for each of them. Along with the savings, with these instances we make sure that we will have availability through all the period that we paid for.
* **On demand instances:** these are the most common type of instances. You only pay for what you use, allowing easy allocation and deallocation of resources, depending on your capacity requirements. Customers are billed at the end of each month.
* **Spot instances:** This is a very interesting concept. In order to achieve a better utilization of their infrastructure, Amazon allows us to bid on unused EC2 capacity and run instances until the current spot instance price exceeds our bid. Amazon based on the available capacity sets the spot price and load of their systems and it is updated in a 5-minute period. The prices of these instances are much lower than what you pay for On-demand instances. As a drawback, the availability of you instances is only assured while the spot price is under bid. As previously stated, Amazon automatically terminates those instances whose bid is exceeded by the spot price. Besides, one cannot stop a spot instance and use it later as it happens with on-demand or reserved instances. Spot instances can only be terminated or rebooted.

Among these types, the spot instances seem to be the most appropriate for running short-term applications under certain conditions, since they provide the same capacity and features as the other instances at a lower rate. These include scientific applications, which usually run for a predictable amount of time, lowering the costs per experiment.

## Many Task Computing (MTC)

Many-Task Computing (MTC) was introduced by Raicu et al. [11][12] in 2008 to describe a class of applications that did not fit easily into the categories of traditional high-performance computing (HPC) or high-throughput computing (HTC). Many MTC applications are structured as graphs of discrete tasks, with explicit input and output dependencies forming the graph edges. In many cases, the data dependencies will be files that are written to and read from a file system shared between the compute resources; however, MTC does not exclude applications in which tasks communicate in other manners.

MTC applications often demand a short time to solution, may be communication intensive or data intensive, and may comprise of a large number of short tasks. Tasks may be small or large, uniprocessor or multiprocessor, compute-intensive or data-intensive. The set of tasks may be static or dynamic, homogeneous or heterogeneous, loosely coupled or tightly coupled. The aggregate number of tasks, quantity of computing, and volumes of data may be extremely large. For many applications, a graph of distinct tasks is a natural way to conceptualize the computation. Structuring an application in this way also gives increased flexibility. For example, it allows tasks to be run on multiple different supercomputers simultaneously; it simplifies failure recovery and allows the application to continue when nodes fail, if tasks write their results to persistent storage as they finish; and it permits the application to be tested and run on varying numbers of nodes without any rewriting or modification.

The hardware of current and future large-scale HPC systems, with their high degree of parallelism and support for intensive communication, is well suited for achieving low turnaround times with large, intensive MTC applications. The MTC paradigm has been defined and built with the scalability of tomorrow’s systems as a priority and can address many of the HPC shortcomings at extreme scales.

## HazelCast

HazelCast is a distributed in-memory data-grid that provides fast access to large amounts of data distributed across a cluster of machines. HazelCast allows you to easily share and partition your application data across your cluster. HazelCast is a peer-to-peer solution (there is no master node, every node is a peer) so there is no single point of failure. HazelCast is pure Java. JVMs that are running HazelCast will dynamically cluster. Although by default HazelCast will use multicast for discovery, it can also be configured to only use TCP/IP for environments where multicast is not available or preferred. Communication among cluster members is always TCP/IP with Java NIO beauty. Default configuration comes with a single backup so if one node fails, no data will be lost. It is as simple as using java.util.{Queue, Set, List, Map}

HazelCast’s Queue service is used as the queuing component in our implementation of CloudKon. It provides reliable and persistent data storage. HazelCast distributed queue is an implementation of java.util.concurrent. BlockingQueue. HazelCast allows you to load and store the distributed queue entries from/to a persistent datastore such as relational database via a queue-store. If queue store is enabled then each entry added to queue will also be stored to configured queue store. When the number of items in queue exceed the memory limit, items will only persisted to queue store, they will not stored in queue memory. Here the queue store configuration options:

* **Binary**: By default, HazelCast stores queue items in serialized form in memory and before inserting into datastore deserializes them.
* **Memory Limit**: This is the number of items after which HazelCast will just store items to datastore. For example if memory limit is 1000, then 1001st item will be just put into datastore. This feature is useful when you want to avoid out-of-memory conditions.
* **Bulk Load**: At initialization of queue, items are loaded from QueueStore in bulks. Bulk load is the size these bulks

HazelCast is highly fault tolerant. It handles failure of nodes gracefully and the data can be replicated across the nodes by specifying the replication factor. HazelCast provides a flexible configuration model providing fine-grained control over each aspect.

After continuous evaluation for its fault tolerance and reliability, we chose HazelCast as the distributed queuing component for our execution fabric. The intent behind choosing HazelCast over Amazon’s distributed queue SQS are

* Fine Grained configuration control
* Support for complex types – object store[In SQS only string support]
* Control on resource allocation and pricing model[Abstraction in case of SQS]
* Flexibility to deploy on private clouds and clusters
* Follows FIFO policy – Message delivery is guaranteed to be in insertion order.
* In Memory Processing delivering efficient speeds
* Highly fault tolerant even in case of multiple node failures

## Cassandra

Apache Cassandra is a No SQL data Store with scalability and high availability without compromising performance. Linear scalability and proven fault-tolerance on commodity hardware or cloud infrastructure make it the perfect platform for mission-critical data. Cassandra's support for replicating across multiple datacenters is best in class, providing lower latency for users and the ability to withstand failures.

Cassandra's data model offers the convenience of column indexes with the performance of log-structured updates, strong support for denormalization and materialized views, and powerful built-in caching

Cassandra incorporates a number of architectural best practices that affect performance

**Fully distributed**- Every Cassandra machine handles a proportionate share of every activity in the system. With each node the same, Cassandra is far simpler to install and operate. There are also no single points of failure and network bottlenecks.

**Log-structured storage engine**- A log-structured engine that avoids overwrites to turn updates into sequential I/O is essential both on hard disks (HDD) and solid-state disks (SSD). On HDD, because seek penalty is so high; on SSD, to avoid write amplification and disk failure.

**Database Level Locking** - Cassandra uses advanced concurrent structures to provide row-level isolation without locking. Cassandra even eliminated the need for row-level locks for index updates.

**Data Modification** - Cassandra’s storage engine only appends updated data, it never has to re-write or re-read existing data. Thus, updates to a Cassandra row or partition stay fast as your dataset grows.

**Replication** – Control over synchronous or asynchronous replication for each update favoring performance over consistency. Highly available asynchronous operations are optimized with features like Hinted Handoff and Read Repair.

**Scalability** - Read and write throughput both increase linearly as new machines are added, with no downtime or interruption to applications.

**Fault** **Tolerant** - Data is automatically replicated to multiple nodes for fault-tolerance. Replication across multiple data centers is supported. Failed nodes can be replaced with no downtime

Cassandra provides an efficient storage mechanism to store the snapshot of the CloudKon execution environment. It provides a simplified interface for persistence of frequent snapshots of the system.

# Design and Implementation of CloudKon

## Design Tenets

The key design principles involved in our implementation of CloudKon are

**Component Based Design –** Our implementation of CloudKon involves components such as client, worker, dynamic dispatcher and monitor. Our design is structured in such a way that all the components are loosely coupled. Unavailability of a particular component does not affect the overall execution of the system. Monitoring component, Client, Worker and Dynamic Dispatcher interact via the distributed queuing service. The endpoints of the various components are also configuration driven to ensure flexible execution even in case of complete failure of a component, The CloudKon processing overhead is very low due to efficient cloud services. The client and worker components do not have a heavy program to run. Many parts of their program calls are the calls to the cloud services, so they are being processed on the third party services. Having totally independent workers and clients, CloudKon does not need to keep any information of its nodes such as the IP address or any other state of its nodes.

**Pull Approach -**  Traditional Job Management system have advocated a centralized scheduler, which schedules tasks on the worker nodes. Here tasks are being posted to the scheduler. This introduces a huge overhead of scheduling decision in the scheduler component. We propose to use a pull approach where each worker pulls the tasks from the queue once it is ready for execution. It distributes the decision making role from one central node to all of the workers in the system. It also requires less communication than the pushing. In the pushing approach the decision maker has to communicate with the workers periodically to update their status as well as distributing the jobs among the workers. On pulling approach the only communication required is pulling the jobs. This approach provides for efficient load balancing across the worker nodes.

**Dual Queuing –** Our implementation of CloudKon utilizes two levels of queuing –one local to the client and another a distributed queue. The intent behind utilizing a twofold model is to ensure coupling of execution of client tasks with a particular worker, i.e. the worker tries to execute all the tasks of a particular client before polling the distributed queue for work from other clients

## Architecture

This section explains about the system design of CloudKon with explanation on individual components involved in the design of CloudKon. All the components are deployed across Amazon EC2 instance.

Figure 1. CloudKon Architecture

The individual components of CloudKon are explained as follows

**Client –** Client is responsible for scheduling tasks to the execution fabric of CloudKon. Client maintains two local queues one for request and another for response respectively. Local Queue service is provisioned via HazleCast. We also evaluated the queue service via ActiveMq but found HazleCast to be more flexible during our evaluations.

Client starts reading tasks from the file or via interactive input and starts queuing the tasks to local queue. Each task is an executable java object, which is converted as object message and queued. Here we explored the possibility of batching a configurable number of tasks equal to the worker threads. Considering the worker threads as 10, we bundled 10 tasks into a single task and scheduled to the local queue. Once all the tasks are queued in the request, Client advertises task messages in the distributed queue. The number of advertising messages is computed by the formula NT/NW where NW is the number of workers and NT is the number of tasks.

In this model during execution of multiple clients, client starvation occurred since all workers were busy executing the workload of previously scheduled clients. This led to us a configurable parameter for advertising tasks. Each client starts advertising the task when 10% [Configurable] of its tasks are queued in the local queue. This ensured that each client’s task is executed as soon as it is queued.

Once all tasks have been posted to the client queue, client starts a response thread, which periodically polls the response queue for the execution results. Since it polls the local queue, communication costs are low. On a configurable period, client checks if all its tasks have been executed failing which it starts advertising one more message to the distributed queue. Client logs the result execution to a output file for examination by users. Client is multithreaded to provide efficient simultaneous execution of all the activities

**Worker –** Worker is the execution part of our implementation. During initialization of the worker, it fetches the distributed queue details from the configuration. After initialization of worker, it starts polling the distributed queue. Worker fetches the first advertised message from the queue. Based on the advertisement it connects to the appropriate client request queue. Worker starts dequeuing each task from the request queue.

Worker is intelligent enough to identify the batched tasks and submits each task to the executor pool. The executor pool holds all the tasks for execution within the worker node and it is a configurable parameter, which corresponds to the worker threads. Once the configured amount of jobs is completed, workers start batching the response to a single object, which is posted to the response queue.

Worker continuously polls the client queue till all the jobs in the client queue are executed. When all the jobs are complete, worker starts polling the distributed queue for further advertisements. Worker implementation is multithreaded to ensure parallel execution of the various maintenance and execution activities. Each worker thread before executing the thread sends a message to the monitoring component to record the snapshot of the worker.

**Queue Component -**  Distributed queue component serves as the backbone of our implementation. A distributed queue decouples different components of the system. Different components can operate independently with the queue component in the middle to decouple different parts of the framework from each other. That makes our design compact, robust and easily extendable.

HazelCast is a highly scalable service that can provide all of the features required to implement a scalable job scheduling system. Using this service, we can achieve the goal of having a system that perfectly fits in the cloud environment and runs on its resources optimally

HazelCast ensures reliable storage for each queued task. Its high scalability and fault tolerance provides efficient support for our implementation. During the course of our evaluation we had terminated nodes of HazelCast cluster and have observed its failover strategy.

Once the HazelCast cluster is set up, the client can continuously advertise the tasks to the queue which worker acts up on. HazelCast is also implemented as a local queue service i.e. single node cluster within the client.

## Dynamic Provisioning

One of the main goals in the cloud environment is the cost-effectiveness. The affordable cost of the resources is one of the main reasons for the users to approach the cloud environment. Therefore it is very important for this project to keep the costs at the lowest possible rate. In order to achieve the cost-effectiveness we have implemented the dynamic provisioning system. Dynamic provisioner is responsible for assigning and launching new workers to the system in order to keep up with the incoming workload.

Amazon CloudWatch provides monitoring for AWS cloud resources and the applications customers run on AWS. Users can use it to collect and track metrics and react immediately. We had to develop a solution similar to Amazon CloudWatch. The problem with using Cloud Watch in our system is that it is coupled with Amazon stack while our aim is to build a execution framework decoupled as possible from AWS. Therefore we decided to implement our own dynamic provisioner.

The dynamic provisioner takes care of launching new worker instances in case of resource shortage. The application checks the queue length of the distributed request queue periodically and compares the queue length with its previous size. If the difference was more than the allowed threshold, it launches a new instance. Both checking interval and the size threshold are configurable parameters.

In order to use the resources efficiently, we have added a feature to the worker nodes. During worker initialization , a termination thread is kicked off in the background The purpose of termination daemon thread kicked during initialization is to identify if the worker is idle. If worker remains idle during the configurable lease period, it is terminated by sending the termination request. EC2 instances are charged on an hourly basis and hence idle instances are terminated near the lease period by default. We have provided configurable time out period in our system so that in Private cloud environment, the timeout can be configured to a lesser period to ensure better utilization of resources if required.

## Execution Model

The typical execution flow of CloudKon are

* HazelCast cluster is initialized with the participating nodes registering themselves to form the peer to peer network
* Cassandra Cluster is initialized if the monitoring is enabled for the execution
* Initial set of worker nodes are started and start connecting to the Cassandra and HazelCast cluster
* When user needs to submit tasks , clients are initialized with the appropriate tasks
* Client starts submitting tasks to the local Queue
* Client also starts advertising the local queue details in distributed HazelCast Cluster
* Free worker nodes pick the queue details from distributed queue and connect to the local client queue
* Once the tasks are completed, worker starts posting results to local response queue of client
* Client collates the results from response queue and once all the results are obtained terminates its execution
* In Dynamic provisioning, dynamic component periodically monitors the queue length and if it exceeds the threshold initiates new workers
* Each worker when reaching its self terminating period terminates itself

## Monitoring

The modified CloudKon provides detailed monitoring abilities. We have structures to monitor and report the following matrixes.

* The Wait Queue Details: This structure measures, stores and reports the Wait Queue Length the time it was recorded (nanosecond). This is done using an atomic integer and the incremented and decremented by clients and workers respectively. The value of this atomic integer is recorded by a thread which runs on client machine. The time when the measurement is taken is stored in nanoseconds.
* The CPU utilization of workers (CPU utilization /minute): This is done by using the amazon services; since measuring the system performance from within a EC2 node is not accurate and the recommended approach for monitoring the CPU Utilization is to use the monitoring services of Amazon. Unfortunately the finest granularity provided by this service is at per minute limit. This monitor can be easily extended to monitor the Disk read write performance, network utilization Ram utilization etc…
* The Client status details: This structure allows the user to monitor and measure the start time, end time and the time it finishes publishing all its jobs in its local queue. This information is critical to calculate the throughput of the system and to figure out the bottlenecks related to slow client job submit rates. The time when the measurement is taken is stored in nanoseconds.
* The Per Client Response Queue details: This structure allows us to see how a single client’s Response Queue behaved during the execution of the entire system. This matrix allows the user to see if there was any starvation of clients and see if there are any bottle necks in the system. This stores the response queue length and the nanosecond when the information was recorded.
* The throughput details per client: Though we can measure the through put of the entire system by using the details in the client status details; if we need to figure out how the system through put behaved during the complete duration of the experiment we need to store the through put of individual clients at regular intervals. This component does the same and allows us to monitor the details related to through put per client. The time when the measurement is taken is stored in nanoseconds.
* The number or worker threads details: When the system runs in dynamic mode; we need this measurement to figure out how many workers were on per stage; since workers automatically terminate after a specific time out interval. This is the matrix reports the number of workers running per stage. This allows us to check the performance of dynamic provisioning; and see how the system behaved when the advertisement queue size increases. The time when the measurement is taken is stored in nanoseconds.
* Worker status details: During dynamic provisioning we need this information to check how the worker behaved during the run at different time periods. This matrix stores the status of worker (busy or free) at the time the measurement is taken. The time when the measurement is taken is stored in nanoseconds.

The CPU seconds wasted/used measurement: This is calculated via two atomic integer of Hazelcast and is useful to understand the resource utilization of the system when the system is working in dynamic provisioning mode. The worker monitoring thread is used to record the worker free/busy status is re-used for this to avoid unnecessary overheads.

We have used a Cassandra cluster to store the monitoring information which is used for reporting and analysis where the consistency is not a huge concern and used hazel cast to monitor and record information that has to be consistent and accurate. This allows us to do in-depth analysis of the system and figure out the bottlenecks and resolve them faster.

The monitoring system is available and scalable easily; since both Cassandra and Hazelcast clusters can scale up or down vertically as well as horizontally seamlessly by simply adding or shutting down nodes respectievely. The bulk of the monitoring is done using separate threads; to minimize the impact on critical activities. We can configure the poll times of each of these monitors separately using the configuration in order to further tweak the monitoring engine.

## Implementation Details

We have implemented all of the CloudKon components in Java. Our implementation is multithreaded in both Client and Worker component codes. We have used some open source libraries in our implementation. The libraries include:

* AWS Java SDK library, for communicating with different AWS services [13]
* HazelCast library for queuing and dequeuing tasks[14]
* Cassandra library for storing and retrieving the system snapshot[15]
* Cassandra columnar database engine[15]
* Dependent open source libraries

### Configuration

Our implementation supports a versatile configuration driven application. Many of the features in both of these systems such as Monitoring, Consistency, number of threads and message packing size can be enabled, disabled or modified as input argument of the program. The configurable parameters are listed below

* numofWorkerThreads – Number of threads inside a remote worker for multithreaded execution
* clientPollTime – Period till which client waits for results from execution system. After the period, client advertises again in the distributed queue
* Task Info – Task count, length and nature of the task is configured here. If task type is I/O file size and location of the file can be specified
* MonitoringEnabled – If set to true, monitoring information are collected in Cassandra engine.
* througputPollTime – Polling period to retrieve snapshot information
* workerSelfTermEnabled – Automatic termination of the worker which would be enabled in public cloud environment
* numOfWorkers – Number of workers to be initialized during the initiation of cloud

# EVALUVATION AND OBSERVATIONS

This section describes the various evaluations done on the modified CloudKon to check its availability, scalability, and performance. In all our experiments we used m1.medium instances of Amazon EC2. We have run all of our experiments on us.east.1 datacenter of Amazon. All of the instances had Linux Operating Systems. Our framework works on any OS that has a JRE 1. 6 (or above) running on it. We have used bash scripting language to semi automate the experiments. This is the same set up that was used in original CloudKon’s evaluation.

## Throughput and Scalability

In order to measure the throughput and latency of our system we run sleep 0 tasks on worker nodes. We have evaluated the performance of CloudKon on multiple instances, starting from 1 instance and extending the experiment up to 64 instances. There are 2 client threads and 4 worker threads running on each instance. Each instance submits 16000 tasks. On the largest scale (64 instances) our system runs 1024000 tasks on each experiment.

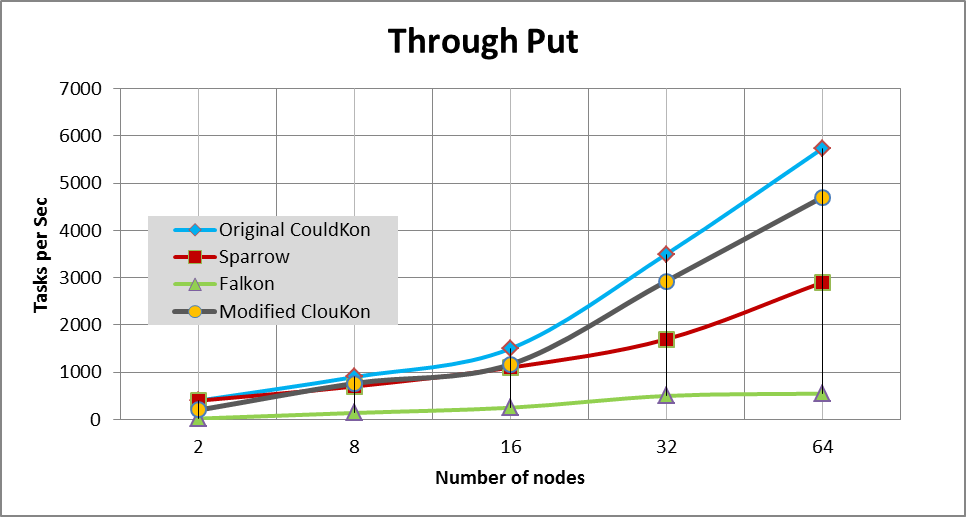


Figure 1. Comparing the throughput of different JobExecution systems.

.

We evaluated the throughput of modified CloudKon just the way Original CloudKon was evaluated i.e. we scaled it from 1 to 64 instances running 16000 to 1024000 tasks. The results show that Modified CloudKon achieves almost linear speedup starting from 235 tasks per second on 2 instances to 4704 tasks per second on 64 instances. Therefore we predict that our solution scales at the same rate on larger scales.

We can observer a drop of around 1000 tasks per second in comparison with the Original CloudKon and it’s due to 2 main following reasons.

SQS is managed by Amazon and it scales automatically when the load on the queuing system increases; whereas in out model as of now we don’t have an automatic scaling of Hazelcast cluster enabled. Currently we used m2.xlarge 10 node cluster of for this and this is acting as a bottle neck. A good mixture of m2.2xlarge nodes and c1.xlarge nodes forming the cluster which scales automatically based on load will improve the performance of the system and we are working towards that.

Original CloudKon was submitting tasks via Google protocol buffers. The tasks mainly contained sleep commands and hence this light weight protocol results in efficient use of the system and very high performance. In our modified version we are submitting tasks which are java serializable Thread objects. The overhead involved in serialization (at client end while submitting tasks) and de-serialization (at Worker end) limits the performance of the system. But we believe that this tradeoff is justified since now the system can support complex tasks which can run independently.

The following graph shows a closer analysis of a the system when submitted to heavy load of small task length. 32 clients submitting 62500 tasks of 1 millisecond to 128 executors running on 32 nodes all running on m1.medium instance of Amazon EC2 . A 10 node m2.2xlarge nodes were used as the Distributed queue component and a 2 node m2.2xlarge Cassandra cluster was used as monitoring database.

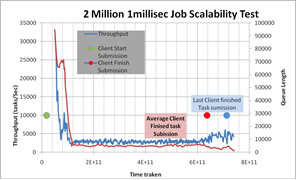


Figure 1. Throughput variation during system execution (2million 1 mill sec jobs)

.

We can clearly see the average through put above 3000 task per second (3527 tasks per sec) which is at par with the through put of original CloudKon. We can clearly see a fall of through put in the initial stages and then a stabilized through put and later a increase in through put towards the end. The very high through put at the start is because all workers are able to retrieve tasks from already filled local task queues of clients (clients advertise them self only when there are 10% tasks in their local queues).The stabilized state of throughout is because of the bottleneck caused by slow speed of client submissions; i.e. clients are still busy submitting tasks in their local queue delaying the workers from pulling the jobs. This becomes clear as we can see a jump in through put towards the end when clients have finally finished all their submissions. The red dot indicates when the most of the client finished submitting all their tasks.

To further measure how the system behaves when there are tasks in Wait queue we increased the job lengths to 1 second.

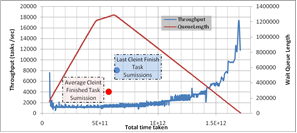


Figure 1. Throughput variation during system execution (2million 1 second jobs)

.

Since the task length is higher this time we can see that the through put is not high during the start phase in comparison with the previous experiment observation; and it is quickly stabilized. The stabilization is mainly because of the task length and we see the trend continue till clients finish submitting their tasks in the queue.

Red dot marks the time when most of the clients have finished their submissions and blue dot indicates the time when the last clients completes the task submission phase. It can be seen that there is a incremental increase in through put after this point. This is due to the fact that the Hazelcast queue component is freed form servicing the client submissions ; which is 20% of the total Hazelcast clients (32/160) and is only servicing the workers. The access is always of a "take" from the queue and no further modifications (puts) are made to this queue. Since Hazelcast cluster is a consistent distributed store; this frees it from a lot of internal overheads associated with cluster synchronization.

The next major jump happens after the yellow dot; which indicates the completion of half of the clients. We attribute this again to the lessened load on distributed Hazelcast cluster; because by now it needs to service only 40% of its total load. As more and more clients finish the load on the cluster decreases and we can see increase in through put. The added fact that by now most of the workers will be servicing a lower number of clients adds to reasons for the increase in through put.

This strengths our assumption that with an auto scaling Hazelcast queue component we can achieve much higher through put with the same model. We intend to study this further.

In comparison to original CloudKon our model does not have the issue of duplicate task controller. Hazelcast uses a Concurrent Distributed Queue and provides us with exactly once delivery of tasks unlike SQS.

We measured the through put of our model while doing File IO operations of different sizes. We did this in order to check how the system behaves when there is IO bound tasks involved.

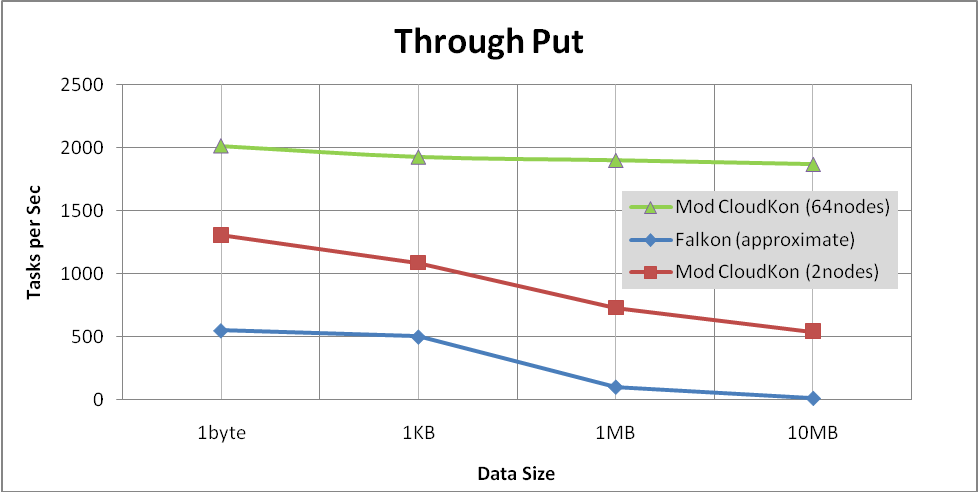


Figure 1. Through put during IO operation

.

We can see a drop of 50% in through put in comparison with sleep 0 tasks; which is mainly due to the involved file IO. But subsequently we can see that the through put is not degrading as much as it was for Falkon (the data for Falkon is approximated from the reference []). This is mainly because due to the centralized model of Falkon limiting its through put and because of the optimized EBS volumes CloudKon uses for writing.

## Efficiency of modified CloudKon

Efficiency of the system dictates its ability to run short tasks with less than a second task length efficiently. The following graph shows the efficiency of the system for various task lengths.

We can see that the system nears optimal efficiency of 1 when the task lengths are high. We can see a clear jump in efficiency for the modified version of CloudKon once the task lengths exceed the 32millisecond mark. This is mainly due to the fact that the overhead associated with serialization and de-serialization of the Thread objects (used as tasks) is behaving as a bottle neck for tasks which have length less than 32 milliseconds. Once this threshold is crossed the task length becomes the main component for execution. We can see that the for tasks of length 1 sec we are reaching ~90% efficiency; indicating that modified version is not adding any overhead

Figure 1: Efficiency vs. Task Length Analysis of various task scheduling systems

.

## Impact of Late Advertisements of tasks

As explained in section we have a advertisement based model where each client advertises its own local queue information in a global queue which all the workers listen to. It’s critical that we find a right time for these advertisements to be made; because if we do it too early then we may end up in a scenario where Workers tried to pull tasks from the local queue and the local queue was not filled. This will lead to huge performance loss if workers use up all the advertisements made before the client fills the local queue.

And on the same line if we delay the advertisements; i.e. wait till the clients fill the local Queue with tasks and then do the advertisements; it would lead to wastage of worker nodes utilization. Since a worker will service a client till the tasks from the clients local queue is not completed another issue is that it late advertisements lead to starvation of few clients. As we can see from the image below which shows the start and stop times of 64 clients (32 nodes); few of the clients (green ring ) are started and finished quiet early and some clients (red ring) are serviced only towards the end.

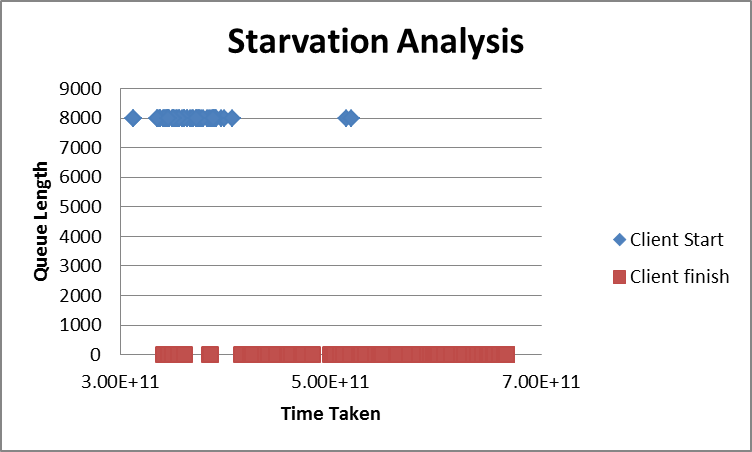


Figure 1: Start and End time analysis of Clients

.

A deeper analysis of the individual queue lengths of these respective clients further establishes this fact. The client that does the advertisement first gets picked initially and finishes early. The starved client which was only able to advertise towards the end has to wait almost till the end of the experiment. The steep fall of the response Queue length is a also an indicator that all the workers are working for this starved client towards the end giving a quick response to the tasks in queue.

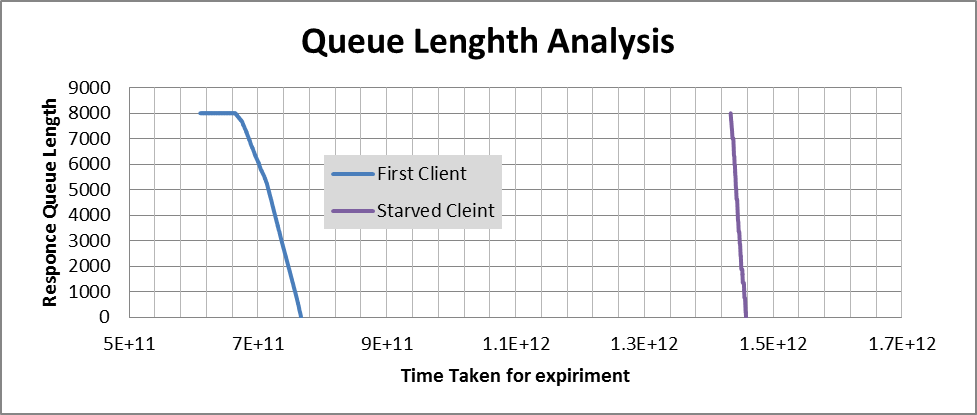


Figure 1: Response Queue Lengths of Clients

.

Hence we implemented a staggered advertisement model where the client does the advertisement of local queue information once the client has filled some percentage (configurable) of tasks in its local queue. The following graph shows the result of this implementation where the clients did their advertisements once 10% of total tasks to be submitted are put in local Queue. There is a clear closer grouping of client starts and finishing timings; indicating a fair scheduling policy.

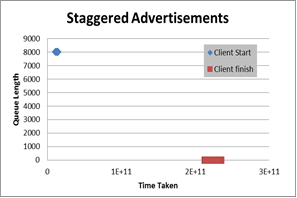


Figure 1: Start and End time analysis of Clients

.

Though our model provides the user to configure this percentage value it’s highly recommended that the user takes a reasonable approach in setting this field. I.e. User can choose a very safer advertisement policy i.e. by increasing this percentage value (80-90%); but then he risks potential starvation and resource wastage. On the other hand a very low percentage (0-10%) set can result in a bigger problem where the workers use up the advertisements made by the client (while the client was filling up its local queue), resulting in a really slow recovering system. As explained in Section X ; we have an automatic recovery mechanism where the clients automatically redo advertisements when a time out happens and there are jobs in its request Queue

## Availability

CloudKon had the inbuilt support of SQS which was a highly available queuing service which we have replaced with Hazel cast. We used 5 m2.xlarge machines to make the Hazelcast cluster which was our distributed queue component. We tested the availably of this service under a load of 64000 tasks and 4 nodes. We incrementally terminated 4/5 nodes forming the Hazelcast cluster while the clients and workers were executing and found that there was no loss of consistency (i.e. all the clients got the result for their submitted jobs) even with a loss of 80% of nodes. We attribute this behavior to the configurable replication feature available in Hazelcast; which allows us to set the nodes holding duplicate copies. We used a replication factor of 1 which means that the data will be backed up by one more node.

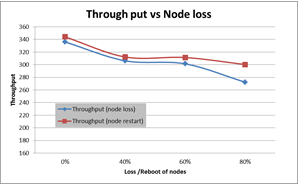


Figure 1: Through put degradation when node loss happens for Hazelcast Queue component.

.

As the figure represents there was a loss of throughput when the cluster had lesser number of nodes. This is because our system effectively uses all the nodes available in the Hazelcast cluster and balances the load in order to achieve optimal performance; and when the nodes in the cluster is less the system performance is bound by the performance of these nodes. A similar study was conducted by rebooting the nodes and we saw no consistency loss and a degradation of performance. The degradation in performance is due to the overhead involved for rejoining of the nodes to the cluster, re-distribution of data, re-allocation of load etc… internal to Hazelcast component.

We also needed to make sure that the Cassandra cluster we use for storing monitoring information was not a bottleneck in the system. We terminated the nodes forming the Cassandra cluster and watch the impact on performance. We tested the availably of this service under a load of 64000 tasks and 4 nodes. We did not find any loss in consistency or degradation in performance. This is due to the fact that the Cassandra cluster is used only to store monitoring information and has no impact on client or executor operations. We use a multi-threaded model for monitoring and separate threads are allocated to do the asynchronous monitoring; so that the critical tasks (job addition, execution, completion etc...) are not impacted due to monitoring.

## Impact of Resource allocation/utilization strategies

In order to study how the system behaves with the home grown dynamic provitioner ; we created a synthetic 5 stage work load.

|  |  |  |
| --- | --- | --- |
| Stage | Number of Tasks | Time per task  (millisec ) |
| 1 | **10** | **120000** |
| 2 | **100** | **180000** |
| 3 | **19** | **120000** |
| 4 | **40** | **120000** |
| 5 | **10** | **120000** |

Figure 1: Through put degradation when node loss happens for Hazelcast Queue component.

.

Please not that we have heterogeneous stages with Stages with varying numbers of tasks and tank length. The following graph plots the number of tasks and number of allocated machines at the end of each stage when we used a 60 sec time out for workers. Once the time out happen the worker nodes terminate on them self.

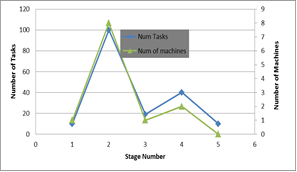
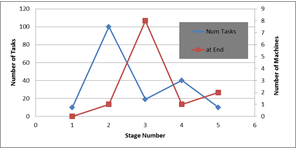


Figure 1: Through put degradation when node loss happens for Hazelcast Queue component.

.

The above graph indicates a good dispatcher mechanism and is in expected lines. As the number of tasks increased in the system; as per stage change, we can see that the system allocated the required nodes. However the when we plotted the same graph for number of nodes at the start of each stage we found that this was way behind the ideal case

.



We can see that the nodes are not allocated immediately once the stage starts. This is mainly because of Amazon EC2 services taking almost 3.44 minutes to start the worker nodes. We did more experiments with different sets of worker termination idle time outs. We define resource utilization and execution efficiency as follows:

*resource\_utilization = ( resource\_used ) / ( resource\_used + resource\_wasted )*

*exec\_efficiency = ideal\_time /actual\_time*

The following table shows ;for each of the time out stratagies the time to complete,resource utilization,execution efficiency,resource allocation.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | **CloudKon 60** | **CloudKon 120** | **CloudKon 180** | **CloudKon 240** |
| **Time to complete (min)** | 24.54 | 23.88 | 23.79 | 23.31 |
| **Resource Utilization (%)** | 75.98 | 47.57 | 26.42 | 13.82 |
| **Execution efficiency (%)** | 18.66 | 19.17 | 19.25 | 19.64 |
| **Resource Allocation** | 13 | 13 | 12 | 11 |

We can see that for CloudKon 240 where the executors had 240 sec set as its idle time out we have less resources allocated and higher execution efficiency but at the same time it has the lowest resource utilization. This is because the system has more workers online at the time of execution and is not spending time to acquire more resources; hence it has better execution efficiency. Whereas in the case of CloudKon 60 it spends more time acquiring the resources (13 resources were allocated and acquired and released) and this impacts the execution efficiency. It has higher resource utilization because the resources were used once they got acquired and were released quickly once there was no work for it to be allocated.

This is in line with the Falkon’s Dynamic resource utilization principles i.e.: The higher the desired resource utilization, the longer the elapsed execution time. This ability to trade off resource utilization and execution efficiency is an advantage of CloudKon. The added advantage that CloudKon has is that we can between spot requests and on demand launch, and from micro instance to c1.xlarge nodes. For high end systems the user will have to pay more but it will provide better efficiency; and low end micro instances will allow the user to run the jobs for lower costs and lower efficiency. This further allows the user to choose between efficiency vs. cost.

# RELATED WORK

# The job schedulers could be centralized, where a single dispatcher manages the job submission, and job execution state updates; or hierarchical, where several dispatchers are organized in a tree-based topology; or distributed, where each computing node maintains its own job execution framework.

# The University of Wisconsin developed one of the earliest job schedulers, Condor [17], to harness the unused CPU cycles on workstations for long-running batch jobs. Slurm [16] is a resource manager designed for Linux clusters of all sizes. It allocates exclusive and/or non-exclusive access to resources to users for some duration of time so they can perform work, and provides a framework for starting, executing, and monitoring work on a set of allocated nodes. Portable Batch System (PBS) [18]was originally developed at NASA Ames to address the needs of HPC, which is a highly configurable product that manages batch and inter-active jobs, and adds the ability to signal, rerun and alter jobs. LSF Batch [22] is the load-sharing and batch-queuing component of a set of workload management tools from Platform Computing of Toronto.

# All these systems target as the HPC or HTC applications, and lack the granularity of scheduling jobs at node/core level, making them hard to be applied to the MTC applications. What’s more, the centralized dispatcher in these systems suffers scalability and reliability issues. In 2007, a light-weight task execution framework, called Falkon [20] was developed. Falkon also has a centralized architecture, and although it scaled and performed magnitude orders better than the state of the art, its centralized architecture will not even scale to petascale systems [19]. A hierarchical implementation of Falkon was shown to scale to a petascale system in [19], the approach taken by Falkon suffered from poor load balancing under failures or unpredictable task execution times.

In this context of job schedulers , CloudKon[21] was developed on Amazon stack to effectively handle MTC workloads. Our work was largely motivated by CloudKon and is much similar with extensions

Table 1. Feature Comparison between CloudKon and our implementation

|  |  |  |
| --- | --- | --- |
| **Feature** | **CloudKon[AWS]** | **CloudKon[Open Source]** |
| AWS Dependency | Tightly coupled with AWS | Loosely coupled with AWS |
| Cluster Configuration | No, it utilizes SQS as the messaging backbone | Tied to HazelCast Cluster for messaging |
| Configuration control | Limited by the components of AWS | Unlimited control with extensible framework |
| Private Cloud | Difficult to decouple from public cloud | Easily portable to private and public cloud |
| Initial Setup | Minimal | Moderate[Cluster configuration] |
| Message Support | String . Objects must be serialized and base 64 encoded to be stored in SQS | Rich support for all types |
| Security | AWS provides good security | In HazelCast security comes  at a price. |
| Duplicate Task Execution | SQS guarantees delivery at least once. Hence a duplicate execution controller is required | Our dual queuing and coupling of worker to client design avoids this repeated task execution |

# CONCLUSION AND FUTURE WORK

Large scale distributed systems require efficient job management system to achieve high throughput and effective system utilization. It is important for the scheduling system to provide high throughput on larger scales and add minimal overhead as possible. CloudKon is a Cloud enabled distributed task execution framework that runs on HazelCast as distributed queue. The evaluation of the CloudKon shows that it is able to provide a very high throughput outperforming other scheduling systems like Sparrow and Falkon. Up to the scale of 64 instances, CloudKon has an almost ideal speed up that shows us that it can easily scale to larger number of instances. The efficiency results show that we can expect high efficiency for the tasks that take hundreds of milliseconds or more.

This work has many directions on its future work. One of the future works is to explore data diffusion[23] techniques to diffuse the required data among worker nodes and utilize data aware scheduling . Another direction is to handle worker failures and resume work from execution point by utilizing a centralized state store as in sparrow[24]. Another direction is to serialize huge execution jars using Google Protocol Buffer[25] to support execution of large tasks. We are also planning to evaluate the execution fabric over a private cloud to monitor its performance. Future direction involves running CloudKon below a workflow engine such as Swift[26] which supports complex workflows. Another idea is to implement resource stealing scheme implemented in Matrix[27] to aim towards tight coupling for handling HPC workloads

# REFERENCES

1. P. Kogge, et. al., “Exascale computing study: Technology challenges in achieving exascale systems,” 2008
2. IEEE Workshop on Many-Task Computing on Grids and Supercomputers (MTAGS08)
3. HazelCast [online] <http://www.hazelcast.com/>
4. Cassandra [online] <http://cassandra.apache.org/>
5. L. Ramakrishnan, R. S. Canon, K. Muriki, I.Sakrejda, and N. J. Wright. Evaluating Interconnect and virtualization performance for high performance computing, ACM Performance Evaluation Review, 40(2), 2012.
6. Q. He, S. Zhou, B. Kobler, D. Duffy, and T.McGlynn. “Case study for running HPC applications in public clouds,” In Proc. of ACM Symposium on High Performance Distributed Computing, 2010.
7. Piyush Mehrotra, Jahed Djomehri, Steve Heistand, Robert Hood, Haoqiang Jin, Arthur Lazanoff, Subhash Saini, and Rupak Biswas. 2012. Performance evaluation of Amazon EC2 for NASA HPC applications. In Proceedings of the 3rd workshop on Scientific Cloud Computing (ScienceCloud '12). ACM, New York, NY, USA, pp. 41-50
8. Guohui Wang and T. S. Eugene Ng. The Impact of Virtualization on Network Performance of Amazon EC2 Data Center. In IEEE INFOCOM, 2010
9. Amazon Elastic Compute Cloud (Amazon EC2), Amazon Web Services, [online] 2013, (<http://aws.amazon.com/ec2/>)
10. Xen Hypervisor, xen.org, [online] 2013, (<http://www.xen.org/products/xenhyp.html>)
11. I. Raicu, Y. Zhao, I. Foster, “Many-Task Computing for Grids and Supercomputers,” 1st IEEE Workshop on Many-Task Computing on Grids and Supercomputers (MTAGS) 2008.
12. I. Raicu. "Many-Task Computing: Bridging the Gap between High Throughput Computing and High Performance Computing", Computer Science Dept.,University of Chicago, Doctorate Dissertation, March 2009
13. AWS SDK for Java, Amazon Web Services,[online] 2013, <http://aws.amazon.com/sdkforjava>
14. HazelCast Library for Java, [online] <http://www.hazelcast.com/downloads.jsp>
15. Cassandra Library for Java and database engine, [online] <http://cassandra.apache.org/download/>
16. M. A. Jette et. al, Slurm: Simple linux utility for resource management. In In Lecture Notes in Computer Sicence: Proceedings of Job Scheduling Strategies for Prarallel Procesing (JSSPP) 2003 (2002), Springer-Verlag, pp. 44-60.
17. D. Thain, T. Tannenbaum, M. Livny, “Distributed Computing in Practice: The Condor Experience” Concurrency and Computation: Practice and Experience 17 (2-4), pp. 323-356, 2005.
18. B. Bode et. al. “The Portable Batch Scheduler and the Maui Scheduler on Linux Clusters,” Usenix, 4th Annual Linux Showcase & Conference, 2000.
19. I. Raicu, et. al. “Toward Loosely Coupled Programming on Petascale Systems,” IEEE SC 2008.
20. I. Raicu, et. al. “Falkon: A Fast and Light-weight tasK executiON Framework,” IEEE/ACM SC 2007.
21. I Raicu, Iman Sadooghi, “CloudKon: a Cloud enabled Distributed tasK executiON framework”
22. LSF,[online],http://platform.com/Products/TheLSFSuite/Batch, 2012
23. Raicu, et. al. “The Quest for Scalable Support of Data-Intensive Workloads in Distributed Systems” , HPDC09,2009
24. Kay Ousterhout, Patrick Wendell, Matei Zaharia, Ion Stoica “Sparrow: Distributed, Low Latency Scheduling” pp.14-15
25. Google Protocol Buffers [online] <https://developers.google.com/protocol-buffers/>
26. I. Raicu, et. al. “Swift: Fast, Reliable, Loosely Coupled Parallel Computation” , IEEE International Workshop on Scientific Workflows , 2007
27. Ke Wang, Anupam Rajendran, Ioan Raicu, “MATRIX: MAny-Task computing execution fabRIc at eXascale”,Sep 2013