CloudKon: DTS

Distributed Task Scheduling with Amazon STACK

Contents

[Table of Figures 2](#_Toc370111926)

[Team 3](#_Toc370111927)

[Abstract 3](#_Toc370111928)

[Problem Statement 3](#_Toc370111929)

[Related Work 3](#_Toc370111930)

[Proposed Solution 3](#_Toc370111931)

[Design Elements 4](#_Toc370111932)

[Monitoring component. 5](#_Toc370111933)

[Client component: 5](#_Toc370111934)

[Dynamic provisioning: 6](#_Toc370111935)

[Mliestones 7](#_Toc370111936)

[Time lines 7](#_Toc370111937)

[Evaluation Metrics 8](#_Toc370111938)

[MidTerm Progress Report 8](#_Toc370111939)

[Dynamic Hazel cast cluster using in ec2 AMI’s. 8](#_Toc370111940)

[Failed approaches: 8](#_Toc370111941)

[Working Approach 9](#_Toc370111942)

[Evaluation 9](#_Toc370111943)

[Dynamic Cassandra Cluster using in ec2 AMI’s. 9](#_Toc370111944)

[Failed approaches: 9](#_Toc370111945)

[Successful Approach: 10](#_Toc370111946)

[Evaluation 10](#_Toc370111947)

[Monitoring of ec2 worker node’s CPU Utilization and storing it in Dynamic Cassandra Cluster: 10](#_Toc370111948)

[Failed approaches: 10](#_Toc370111949)

[Successful Approach: 10](#_Toc370111950)

[Evaluation: 10](#_Toc370111951)

[Posting Queue Metadata to SQS 10](#_Toc370111952)

[Failed approaches: 11](#_Toc370111953)

[Successful Approach: 11](#_Toc370111954)

[Evaluation: 11](#_Toc370111955)

[Posting and Retrieving tasks from ActiveMQ 11](#_Toc370111956)

[Failed approaches: 11](#_Toc370111957)

[Successful Approach: 11](#_Toc370111958)

[Evaluation: 11](#_Toc370111959)

[MultiThreaded Worker 11](#_Toc370111960)

[Failed approaches: 12](#_Toc370111961)

[Successful Approach: 12](#_Toc370111962)

[Evaluation: 12](#_Toc370111963)

[Future Work 12](#_Toc370111964)

[Evaluation 12](#_Toc370111965)

[Throughput (tasks executed/sec): 12](#_Toc370111966)

[Efficiency and Speedup: 12](#_Toc370111967)

[Scalability 12](#_Toc370111968)

[Dynamic Provisioning: 12](#_Toc370111969)

[Contributions 13](#_Toc370111970)

[References 13](#_Toc370111971)

# Table of Figures

[Figure 1 : Architecture Diagram 4](#_Toc370105379)

# Team

* Kamalnath N.G (CWID : A20260987)
* Rajagopal Parthasarathi(CWID:A20276919)

# Abstract

Many Task computing 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 generally MTC tasks are communication intensive. Task Scheduling of MTC tasks and execution over large scale, distributed systems plays an important role on achieving good performance and high system utilization.The goal of this project is to leverage Amazon stack as a public cloud service to provide a scalable task scheduling system that supports Many Task Computing (MTC) workloads.

# Problem Statement

Predictions are that by the end of this decade, we will have exascale system with millions of nodes and billions of threads of execution. Unfortunately, today’s jobschedulers 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 workloads.

# Related Work

The University of Wisconsin developed one of the earliest job schedulers, Condor [1], to harness the unused CPU cycles on workstations for long-running batch jobs. Slurm [2] is a resource manager designed for Linux clusters of all sizes. Portable Batch System (PBS) [3] 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. In 2007, a light-weight task execution framework, called Falkon [4] 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. Sparrow is another scheduling system that focuses on scheduling very short jobs that complete within hundreds of milliseconds. It has a decentralized architecture that makes it highly scalable.

# Proposed Solution

The goal of this project is to implement a job scheduling/management system that satisfies four major objectives:

* **Scalability -**  Throughput increases with the increase in number of distributed node
* **Load Balancing -**  Efficient and effectiive implementation of system to distribute the workload across all the interacting nodes.
* **Light-weight** - System should involve minimal overhead even while executing fine granular workloads.
* **Loosely Coupled** - Critical Design element to achieve efficient implementation of distributed execution fabric

CloudKon is an effective implementation for distributed task scheduling framework in a cloud based environment. Our proposal is to implement an system similar to CloudKon over Amazon stack. The key differentiators of our solution from CloudKon is explained in the section below

# Design Elements

# 

Figure : Architecture Diagram

## Monitoring component.

We are planning to use Cassandra cluster for this. Cassandra is known for its write performance and scalability. We can always scale the cluster if we feel that the current nodes are not efficiently handling the load. And since Cassandra adopts an eventual/tuneable consistency model we can ensure that the writes for the worker nodes are not blocked.

Since this monitoring information is used for reporting and is not mission critical we can allow some level of inconsistency in the system when reporting it in Real time and assume that the Cassandra’s in built replica synchronization mechanism will ensure the correctness of data during offline reporting done for analysis.

We plan to use the Columnar architecture of Cassandra and arrange the data to fit a model so that the read performance is also high (even with a consistency level (CL)of 2 for every read operation).

## Client component:

We plan to modify the Client component in 2 ways:

1. The client will be posting its tasks in an Internal Client Local Queue (ICLQ). We are planning to use an ActiveMQ for this. The client will then post **Xc** messages in the AmazonSQS.
   1. The messages will contain the client ID and the information regarding connecting to the ICLQ.
   2. **Xc** is calculated as follows:
      1. **Xc = NTc /( NThw\* NITw)**

**NTc** = Number of Tasks posted in Active MQ(ICLQ) by client.

**NThw** = Number of threads per worker nodes.

**NITw**=Number of Iterations the worker will make. This means that the Worker will make N iterations to the same client to pull messages from the same client.

The idea here is to control the number of workers catering to same client; So that we can later use a batch and send the results together at worker end instead of pinging the client every time for results.

The Client will monitor its Response Q for results from the workers. Since Active MQ will ensure that one message is picked by only one worker we don’t need a duplicate checker component.

For fault tolerate (worker nodes crashing with tasks taken form Q) we will have intelligence at client end to check whether responses are received for all tasks after some time out and resend the tasks for completion . Though we don’t expect EC2 instances to go down with tasks we will nonetheless have this component for completeness.

The second change will be to send non trivial commands as tasks from client

There are two approaches considered.

1. Send custom Java Runnable Objects as tasks and use java’s native serialization.
   1. In this approach we plan to constructs objects which implement runnable interface and have custom run implementation.
   2. We then serialize these objects using java serialization and use them as tasks in ICLQ.
2. Second approach is to use Google protocol buffer and send across executable jars.

If we see that Approach 1 is not scaling well we will try to implement approach 2.

The Client will also send the following information to a DHT used for dynamic provisioning.

1. Time each of its tasks will take to complete. (Client ID > Time taken per task **TTac**)

*A client sending heterogeneous tasks varying in execution time is not in scope for this project.*

**Worker threads:**

The worker thread will do the following:

* Poll the Amazon SQS and get **One message.** 
  1. This message will contain the information about which client it will cater to and the ActiveMQ information from where it should pull the tasks
* The worker will the take tasks from active MQ and spawn threads (configured numbers)to handle these tasks.
  1. The threads will execute the task and populate the shared memory ConcurrentHashmap with results (TaskID ->result).
     1. Once this hash map exceeds a certain number we will flush it to the Local Response Q of the client.
* The Worker will iterate **NITw** number of times and repeat step 2
* Once the iteration is over It Flushes all the responses in its Hashmap to client and Goes to Step 1.
* The worker will periodically calculate how much more time it will be servicing its current client and will report the same in DHT. (Worker ID ->TLw)

Total tasks to be executed by worker = **NThw\* NITw**

Task already completed **= NTwc**

Total time to be spent by worker for the client = **((NThw\* NITw)-NTwc)\* TTac.**

* Workers will terminate on idling for more than some configured time and if the Amazon lease is going to expire.

## Dynamic provisioning:

This component will poll the DHT and see how many workers are currently busy.

It will periodically poll SQS to check whether any new Request from workers have come from clients.

If it sees Y messages in SQS then it does the following:

If there is some ( Z) workers who are about to complete their assignment with Client(s) and become free in next 2 minutes (configurable) then it will spawn Y-Z new workers.

This way we will not spawn a lot of workers and underutilize the existing workers. The intent here is that reusing an already running worker will be more beneficial than starting new workers; since

* The startup procedure (bidding, machine ,JVM startup) will take un utilized time.
* Cost of EC2. Since EC2 is charged per hour it makes sense to re-use the existing EC2 instances which are already started.

The wastage of resources (money) and time will increase with the scale; and using the above model of reuse we will improve both the performance and reduce cost. The DHT component we will be using is Hazelcast which was found to auto scale and be fault tolerant and consistent.

# Mliestones

1. **MileStone1. Core functionality**
   * 1. Client with ActiveMQ ,and Fault tolerant.
     2. Messages are sent via SQS
     3. No Dynamic Provisioning
     4. Bag of Tasks Workload with Java Serialization
     5. Monitoring component with Cassandra
2. **MileStone2 : Workers Batching results.**
   * 1. Dynamic Provisioning is included [Not very smart one].
     2. Efficient reuse of “about to finish workers” not implemented.
3. **Milestone 3 : Dymanic Provitioning**
   * 1. Dynamic Provisioning using Hazel cast to enhancing it.
     2. Efficient reuse of “about to finish workers” implemented.

Experimental Milestones

* Milestone 4 - An executable jar as an workload rather than Serializable Java objects
* Milestone 5 - Replace SQS with HazleCast
* Milestone 6- Attempt for other types of workloads- Pipeline

# Time lines

|  |  |  |
| --- | --- | --- |
| Target Milestone | Start Date | End Date |
| MileStone1. Core functionality | 21-09-013 | 18-10-013 |
| MileStone2 : Workers Batching results. | 18-10-013 | 25-10-013 |
| Milestone 3 : Dymanic Provitioning | 25-10-013 | 03-11-013 |
| Benchmarking System | 04-11-013 | 15-11-013 |
| Experimental Milestone1 | 04-11-013 | 15-11-013 |
| Experimented Milestone2,3 | 15-11-013 | 22-11-013 |
| Project Report /Presentation Work | 22-11-013 | 02-12-013 |

# Evaluation Metrics

Throughput, Latency, Efficiency, Utilization of the modified CloudKon will be measured against the original results obtained for CloudKon presented here[5].

# MidTerm Progress Report

## Dynamic Hazel cast cluster using in ec2 AMI’s.

We are planning to use hazel cast cluster in order to replace the SQS component. Since Amazon SQS is highly available, scalable and reliable service we needed the same form the replacement. We also needed the new replacement to be faster and support heterogeneous Objects. So that we can extend the Cloudkon to support more than just sleep tasks. We also wanted to replace the DynamoDB component of Cloudkon and intend to use Hazelcast distributed Maps for storing the data. The dynamic provisioning component will use this Distributed Map for storing the data needed for its operation.

I have successfully created a set up where I can dynamically scale up the cluster y starting new instances of an AMI which has hazel cast installed. This ensures us that when the load increases we can always **scale the cluster up**.

With respect to resource utilization we can always scale down by shutting down the nodes; since hazel cast has inbuilt replication and **fault tolerance**.

Since Hazelcast is an in memory data grid; we are expecting it to provide the **speed** which we are looking for and it promises. Hazel cast also supports **Objects** inside its distributed Queues and Maps which will also provide us with the needed support for complex tasks.

Since the Distributed Queue of Hazelcast in FIFO; this will also help us in future in case we intend to support workflows and not just Bag of tasks.

### Failed approaches:

* I initially ran into firewall related and region related issues which were resolved by using proper settings in the security configurations of ec2 nodes.
* When i used the **<tcp-ip enabled="true">** setting in hazelcast.xml to let the hazelnodes discover each other we ran into the issue that amazon provides the ip addresses to new nodes in a difficult to predict way and realized that i needed a new approach.
* When working with **<aws enabled="true">** setting in hazelcast.xml setting I ran into issues related to <tag-key>type</tag-key>. I had over looked this setting and was not providing tags for my ec2 instances.

### Working Approach

* By working with **<aws enabled="true">** and by correctly tagging the ec2 instances I was able to get the nodes discover each other upon start individual start up. There is a small script which invokes the <hazelcast root>/bin/startserver.sh and this script will be called when the system boots.

### Evaluation

* **Fault tolerance:** I have tested the hazel cast systems fault tolerance by creating 3 nodes and inserting values into various nodes and then shutting 2 nodes at the same time. I saw that the remaining 1 node was able to service the request for data. At the point of writing this our framework was almost complete and we intend to more tests with real tasks, workers and clients.
* **Load testing:** On my laptop (1GB heap for java) I was able to successfully put 1million strings into a single node hazel Queue and retrieve it in a reader successfully using the blocking Q set up. This gives us the confidence that even if we have more objects to hold than the memory available to hazel cast; the system will not crash and will continue to operate. We will be compromising on performance in such a scenario for availability. This is highly unlikely since we will have 4GB allocated to each of the nodes in hazel cast. At the point of writing this our framework was almost complete and we intend to more tests with real tasks, workers and clients

## Dynamic Cassandra Cluster using in ec2 AMI’s.

We needed a DataStore to hold date coming from monitoring component. Since we were planning to replace DynamoDB we needed a data store that is scalable, available and one with provides better write speed than DynamoDB. We chose Cassandra for this since it’s a highly available, scalable and it supports **write heavy applications**.

I have successfully able to create a setup where once we have a seed node running; we can scale the system up just by starting an AMI which has Cassandra installed. This resolves the **scalability** for us. And since Cassandra has in built replication and **fault tolerance** we can always shut down nodes when there is no load in the system. Since Cassndra follows a decentralized model it has no Single point failure related issues and provides high **availability**. Cassandra also has an **eventual consistency model** which enables it to support high amount of **write requests**.

### Failed approaches:

* I ran into issues related to path and the script that was supposed to download the configuration details from a server and start Cassandra on machine startup. Though the solution to this was simple; it took a lot of time to figure out the reason for the issue.
* Few ports needed to be explicitly opened in the security configuration settings; which was missed in my initial attempts.

### Successful Approach:

* Once the path and port related issues were rectified I was able to get the cluster up and scale it by starting AMI’s.

### Evaluation

* **Fault tolerance:** I have tested the Cassandra cluster with 2 nodes. I inserted data in to one node,crashed it and retrieved from other node. The data was found to be replicated. At the point of writing this our monitoring component was completed and framework was also almost complete; we intend to more tests with real tasks, workers and clients

## Monitoring of ec2 worker node’s CPU Utilization and storing it in Dynamic Cassandra Cluster:

The current Cloudkon records how ***active*** the system has been using an assumption that the tasks are always sleep jobs. This might not be the case when we are supporting tasks which are not just sleep tasks. Hence we planned to provide CPU utilization /minute of the ec2 worker nodes. We are using Amazon cloud watch API to do this and this is considered the correct way of measuring ec2 instances so that we don’t end up with “steal” time related miscalculations.

To be efficient and to follow standards I decided to use the Cassandra latest CQL java driver for the database related operations.

### Failed approaches:

* I ran into issues related to jar version mismatch. AWS cloud watch API wanted HTTPClient.jar in its latest version and the Cassandra CQL driver had another older version of the jar. This created a strange issue where when the program invoked from the IDE ran without any issue; but when Ran using a java –jar option run into “Content has already been Consumed” exception. It took me some time to figure out that the issue was actually related to a jar mismatch.

### Successful Approach:

* Once the jars were replaced with latest versions I was able to get the component working and we are able to record the CPU utilization of workers using this component.

### Evaluation:

I have run this component on a dummy worker node and recorded its utilization in the Cassandra cluster.

# Posting Queue Metadata to SQS

The key component of CloudKon is posting Messages to the distributed queue. For this purpose initially we are rolling out with Amazon SQS. SQS is a highly reliable distributed offering from Amazon with efficient api support. Our plan was to advertise the client queue details via SQS and workers will pick the messages from SQS to connect to client queue.

### Failed approaches:

* Incompatible jar versions of Amazon SDK and httpclient caused peer Authentication failures when posting messages to SQS
* SQS allowed only String messages. So I planned to serialize the object into bytestream and transfer the message as String. But it resulted in incompatible XML characters being transferred which SQS rejected .

### Successful Approach:

* Once the jars were replaced with latest versions, I was able to post messages to SQS
* Based on AWS forums, we planned to used Base64 encoding to serialize and deserialize the String. But our version of base 64 was causing issues. So we ended up sending the object properties as a string and restoring the object properties at worker end. Since here the queue metadata is only being transferred this approach fits our needs

### Evaluation:

I have tested this both in local node and also on Amazon EC2 instance with expected results

# Posting and Retrieving tasks from ActiveMQ

Our idea was to utilize request and response queues at client side to utilize batching and easier results collation. ActiveMq is a queuing service which provides rich service to consumers and producers. Here ActiveMQ is used as local Queuing provider internal to the client and not as distributed queue provider.

Client posts individual tasks to the Request queue. Worker pulls each task from the request queue and executes the tasks. Once the task results are completed, worker posts results back to the response queue which is polled by the client thread.

### Failed approaches:

* I was able to successfully test ActiveMQ without failures. I had enabled security group settings in EC2 to allow all TCP communications.

### Successful Approach:

* I was able to successfully post tasks to ActiveMQ and retrieve tasks from executor
* I was able to implement batching by combining multiple task results into single task object

### Evaluation:

I have tested this both in local node and also on Amazon EC2 instance with expected results. For my testing I had posted 1000 tasks with various sleep times. Further evaluation has to be conducted with multiple worker nodes.

# MultiThreaded Worker

Multthreaded worker forms the execution fabric. Once the worker is initialized it picks up task metadata from distributed queue. A task collating thread along with termination timer is kicked off parallelly. With metadata information tasks are pulled from the corresponding client request queues and tasks are executed in multithreaded fashion. Once the worker executes each job,results are accumulated locally. The collating thread accumulates the result once task results reach a specified number or specified time and posts the batch response to client queue. The multithreading was achieved using Executor Service and Timer task in java.

### Failed approaches:

* Initially I was accumulating messages but still sending messages one by one in response queue.

### Successful Approach:

* Tasks were aggregated into single task object and the single response was posted back to client response queue.

### Evaluation:

I have tested this both in local node and also on Amazon EC2 instance with expected results. For my testing I had posted 1000 tasks with various sleep times. Further evaluation has to be conducted with multiple worker nodes.

# Future Work

* Evaluate HazleCast as an alternative for SQS
* Test the sleep tasks with varying workloads in large scale environment
* Compare the System with similar task execution frameworks

## Evaluation

Throughput (tasks executed/sec): We plan to measured performance running “sleep 0” on a maximum of 300 worker nodes.We also intend to check how IO bound tasks affect the performance with IO operations of varing from 1B to 1GB.

### Efficiency and Speedup:

Efficiency (EP=SP/P) as a function of number of processors (P) and task length; speedup is defined as SP=Tn/TP, where Tn is the execution time on n processors.We plan to measure the efficiency for tasks of varying sleep order (sleep 1 sec to sleep 64)

### Scalability

For test we will have a client that submits two million “sleep 0” tasks to a dispatcher configured with a Java heap size set to 1.5GB we will have 100 executors.

### Dynamic Provisioning:

To study provisioner performance, we will construct a synthetic 18-stage workload, in which the numbers of tasks and task will vary in lengths between stages.We will measure the resource utilization **as RU = resources used /(resource used + resource wasted)**

# Contributions

|  |  |
| --- | --- |
| Item | Team Member |
| Dynamic Hazel cast cluster using in ec2 AMI’s | KamalNath N.G |
| Dynamic Cassandra Cluster using in ec2 AMI’s | KamalNath N.G |
| Monitoring of ec2 worker node’s CPU Utilization and storing it in Dynamic Cassandra Cluster | KamalNath N.G |
| Posting Queue Metadata to SQS | Rajagopal Parthasarathi |
| Posting and Retrieving tasks from ActiveMQ | Rajagopal Parthasarathi |
| MultiThreaded Worker | Rajagopal Parthasarathi |

# References

1. 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.
2. M. A. Jette et. al, Slurm: Simple linux utility for resource management. In Lecture Notes in Computer Sicence: Proceedings of Job Scheduling Strategies for Prarallel Procesing (JSSPP) 2003 (2002), Springer-Verlag, pp. 44-60
3. B. Bode et. al. “The Portable Batch Scheduler and the Maui Scheduler on Linux Clusters,” Usenix, 4th Annual Linux Showcase & Conference, 2000.
4. I. Raicu, et. al. “Falkon: A Fast and Light-weight tasK executiON Framework,” IEEE/ACM SC 2007
5. CloudKon <http://www.cs.iit.edu/~iraicu/teaching/CS554-F13/proj/CloudKon-DTS.pdf>
6. [ZHT: A Light-weight Reliable Persistent Dynamic Scalable Zero-hop Distributed Hash Table, IPDPS 2013](http://datasys.cs.iit.edu/projects/ZHT/ZHT-CRC-PID2666213-Final.pdf)
7. Hazlecast : <http://www.hazelcast.com/>