CloudKon: DTS

Distributed Task Scheduling with Amazon STACK

Contents

[Team 2](#_Toc368250219)

[Abstract 2](#_Toc368250220)

[Problem Statement 2](#_Toc368250221)

[Related Work 2](#_Toc368250222)

[Proposed Solution 2](#_Toc368250223)

[Design modifications 3](#_Toc368250224)

[Monitoring component. 4](#_Toc368250225)

[Client component: 4](#_Toc368250226)

[Dynamic provisioning: 5](#_Toc368250227)

[Mliestones 6](#_Toc368250228)

[Time lines 6](#_Toc368250229)

[Evaluation Metrics 7](#_Toc368250230)

[References 7](#_Toc368250231)

# 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 modifications

Writes **Xc** message (Client ID and ActiveMQ information)

Internal ActiveMQ- Request

Amazon SQS

Reads Tasks

Write Tasks

Reads **One** Message

**Batch** Write Results

Internal ActiveMQ Responce

Writes Task Length

Writes CPU load/Sec

Writes time till Free

Read Results

Monitoring Component

Read system state

Dynamic Positioner

Starts the workers

## 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].

# 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/>