# CloudKon: a Cloud enabled Distributed tasK executiON framework

Iman Sadooghi, Ioan Raicu
isadoogh@iit.edu, iraicu@cs.iit.edu
*Department of Computer Science, Illinois Institute of Technology, Chicago IL, USA*

***Abstract —*** *Task scheduling and execution over large scale, distributed systems plays an important role on achieving good performance and high system utilization. Job management systems need to support applications (e.g. Many-Task Computing – MTC, MapReduce) with a growing number of tasks with finer granularity due to the explosion of parallelism found in today's hardware which requires techniques such as over-decomposition to deliver good performance. Our goal in this work is to provide a compact, light-weight, scalable, and distributed task execution framework (CloudKon) that builds upon cloud computing building blocks (Amazon EC2, SQS, and DynamoDB). Most of Today's state-of-the-art job execution systems have predominantly Master/Slaves architectures, which have inherent limitations, such as scalability issues at extreme scales and single point of failures. On the other hand distributed job management systems are complex, and employ non-trivial load balancing algorithms to maintain good utilization. CloudKon is a distributed job management system that can support millions of tasks from multiple users delivering over 2X the performance compared to other state-of-the-art systems in terms of throughput – all with a code-base of less than 5%. Although this work was motivated by the support of MTC applications, we will outline the possible support of HPC applications as well.*

**Index Terms—** Cloud Computing, Many-Task Computing, distributed scheduling, task execution framework

## 1 INTRODUCTION

The goal of a job scheduling system is to efficiently manage the distributed computing power of workstations, servers, and supercomputers in order to maximize job throughput and system utilization. With the dramatically increase of the scales of today's distributed systems, it is urgent to develop efficient job schedulers. 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].

Unfortunately, today's schedulers have centralized Master/Slaves architecture (e.g. Slurm [2], Condor [3][4], PBS [5], SGE [6]), where a centralized server is in charge of the resource provisioning and job execution. This architecture has worked well in grid computing scales and coarse granular workloads [7], but it has poor scalability at the extreme scales of petascale systems with fine-granular workloads [8][9]. The solution to this problem is to move to the decentralized architectures that avoid using a single component as a manager. Distributed schedulers are normally implemented in either hierarchical [36] or fully distributed architectures [30] to address the scalability issue. Using new architectures can address the potential single point of failure and improve the overall performance of the system up to a certain level, but issues can arise in distributing the tasks and load balancing among the nodes [25].

The idea of using cloud services for high performance computing has been around for several years, but it has not gained traction primarily due to many issues. Having extensive resources, public clouds could be exploited for executing tasks in extreme scales in a distributed fashion. Our goal in this project is to provide a compact and lightweight distributed task execution framework that runs on the Amazon Elastic Compute Cloud (EC2) [16], by leveraging complex distributed building blocks such as the Amazon Simple Queuing Service (SQS) [18] and the Amazon distributed NoSQL key/value store (DynamoDB) [32].

There have been many research works about utilizing public cloud environment on scientific computing and High Performance Computing (HPC). Most of these works show that cloud was not able to perform well running scientific applications [10][11][12][13]. The problem with these works is they are all trying to exploit the cloud using the same approach as traditional clusters and super computers. Using shared resources and virtualization technology makes public clouds totally different than the traditional HPC systems. Instead of running the same traditional applications on a different infrastructure, we are proposing to use the public cloud service based applications that are highly optimized on cloud environment. Using public clouds like Amazon as a job execution resource could be complex for end-users if it only provided raw Infrastructure as a Service (IaaS) [33]. It would be very useful if users could only login to their node and submit jobs without worrying about the resource management.

Another benefit of the cloud services is that using those services, users can implement relatively complicated systems that are able to serve in larger scales with a very short code base in a short period of time. Our goal is to show evidence that using these services we are able to provide a system that provides high quality service that is on par with the state of the art systems in with a significantly smaller code base.

In this project, we implement a scalable task execution framework on Amazon cloud using different AWS cloud services. The most important component of our system is Amazon Simple Queuing Service (SQS) which acts as a content delivery service for the tasks. Other cloud services are also used in this project. Amazon DynamoDB is another cloud service that is used in this project to provide the exactly once delivery of tasks in the 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. A good example for this type of applications is Many Task Computing (MTC) [14]. MTC applications often demand a short time to solution and may be communication intensive or data intensive [15]. Tasks may be small or large, uniprocessor or multiprocessor, compute-intensive or data-intensive.

As we mentioned above, running jobs in extreme scales is starting to be a challenge for current state of the art job management systems that have centralized architecture. On the other hand, the distributed job management systems have the problem of low utilization because of their poor load balancing strategies.

We propose CloudKon as a job management system that achieves good load balancing and high system utilization. Instead of using trivial techniques such as random sampling or hierarchical system design, CloudKon uses distributed queues to deliver the tasks fairly to the workers without any need to for the system to choose between the nodes. The distributed queue serves as a big pool of tasks that is highly available. As soon as a worker is done with running its tasks, it can choose new tasks from the queue and start running them. The benefit of this approach is that is very simple as well as being highly efficient and scalable. Another benefit of this solution is that different system components loosely coupled to each other. That makes the system highly scalable, robust, and easy to upgrade.

The main contributions of this work are:

1. Design and architect a simple light-weight task execution framework using Amazon Cloud services (EC2, SQS, and DynamoDB)
2. Deliver 2X performance improvement with <5% codebase
3. Performance evaluation up to 64-VMs comparing CloudKon with other state-of-the-art 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 studies the related work in the area of task execution systems. Section 4 discusses about the design and implementation details of CloudKon. Section 5 evaluates the performance of the CloudKon in different aspects using different metrics. Finally section 6 discusses about the limitations of the current work, and covers the future directions of this work.

## 2 BACKGROUND & RELATED WORK

This section covers the necessary background information on Amazon EC2, SQS, DynamoDB, and Many-Task Computing (MTC). It also covers related work to job management systems and light-weight task execution frameworks.

## 2.1 Amazon Elastic Compute Cloud (EC2)

Cloud computing services are categorized in three layers of Infrastructure-as-a-Service (IaaS), Platform-as-a-Service (PaaS) and Software-as-a-Service (SaaS). The focus of this paper 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 (EC2) [16] is an IaaS Cloud that provides a raw infrastructure and the associated middleware. Amazon uses XEN hypervisor [17] 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. Basically, an instance is a running virtual machine on Amazon's cloud platform. Each of these instances is 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 of time, 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. The spot price is set by Amazon based on the available capacity 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.

## 2.2 Amazon SQS

Amazon SQS is a fast distributed message delivery fabric that is highly scalable. It is normally used to decouple different components of a cloud application. It can queue unlimited number of short messages. The maximum size for a message is 256 KB [18].

Messages can be sent and read simultaneously on SQS. When a user receives a message, before removing that message, SQS locks the message in the queue without letting other users access it. This keeps other computers from processing the message simultaneously. If the message processing fails, the lock will expire and the message will be available again. SQS guarantees delivery of each message at least once, and provides highly concurrent access to messages. That also means it does not guarantee the exactly once delivery. That means there could be multiple copies of the same message available to read by different users. It also does not guarantee the order of the messages.

## 2.3 Amazon DynamoDB

DynamoDB is a fast, NoSQL database service that provides users to store and retrieve any amount of data, and serve any level of request traffic. It is fully distributed and highly scalable. It is able to handle large amounts of simultaneous write and read. Like other NoSQL databases, DynamoDB does not provide complex data access queries. It lets users save and access the data using its coordinating key. DynamoDB provides some key features such as atomic read and write on the table which comes really useful for our usage.

## 2.4 Many Task Computing

Many-Task Computing (MTC) was introduced by Raicu et al. [14][15] 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.

## 2.5 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 [3], 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. 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) [5] 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 [19] 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 [9] 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 [8]. A hierarchical implementation of Falkon was shown to scale to a petascale system in [8], the approach taken by Falkon suffered from poor load balancing under failures or unpredictable task execution times.

Although distributed load balancing at extreme scales is likely a more scalable and resilient solution, there are many challenges that must be addressed (e.g. utilization, partitioning). Fully distributed strategies have been proposed, including neighborhood averaging scheme (ACWN) [20][21][22][23]. In [23], several distributed and hierarchical load balancing strategies are studied, such as Sender/Receiver Initiated Diffusion (SID/RID), Gradient Model (GM) and a Hierarchical Balancing Method (HBM). Other hierarchical strategies are explored in [22]. Charm++ [24] supports centralized, hierarchical and distributed load balancing. In [24], the authors present an automatic dynamic hierarchical load balancing method for Charm++, which scales up to 16K-cores on a Sun Constellation supercomputer for a synthetic benchmark.

Sparrow is another scheduling system that focuses on scheduling very short jobs that complete within hundreds of milliseconds [25]. It has a decentralized architecture that makes it highly scalable. It also claims to have a good load balancing strategy with near optimal performance using a randomized sampling approach. It has been used as a building block of other systems.

Work stealing is another approach that has been used at small scales successfully in parallel languages such as Cilk [26], to load balance threads on shared memory parallel machines [27][28][29]. However, the scalability of work stealing has not been well explored on modern large-scale systems. In particular, concerns exist that the randomized nature of work stealing can lead to long idle times and poor scalability on large-scale clusters [29]. The largest studies to date of work stealing have been at thousands of cores scales, showing good to excellent efficiency depending on the workloads [29].

MATRIX is an execution fabric that focuses on running Many Task Computing (MTC) jobs [30]. It uses an adaptive job stealing approach that makes it highly scalable and dynamic. It also supports the execution of complex large-scale workflows, and has been shown to scale to 1K-nodes.

Most of these existing light-weight task execution frameworks have been developed from scratch, resulting in code-bases of tens of thousands of lines of code. This leads to systems which are hard and expensive to maintain, and potentially much harder to evolve once initial prototypes have been completed. This work aims to leverage existing distributed and scalable building blocks to deliver an extremely compact distributed task execution framework while maintaining the same level of performance as the best of breed systems.

## 3 DESIGN AND IMPLEMENTATION OF CLOUDKON

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

1. **Scale:** Offer increasing throughput with larger scales through distributed services
2. **Load Balance:** Offer good load balancing at large scale with heterogeneous workloads
3. **Light-weight:** The system should add minimal overhead even at fine granular workloads
4. **Loosely Coupled:** Critical towards making the system compact and robust

In order the achieve scalability, CloudKon uses SQS which is distributed and highly scalable. As a building block of CloudKon, SQS can upload and download large number of messages simultaneously. Therefore it enables the framework to add more clients and workers to the system without decreasing the per node bandwidth of each individual node. The independency of the workers and clients makes the framework perform well on larger scales. In order to provide other functionalities such as monitoring or task execution consistency, CloudKon also uses cloud services such as DynamoDB that are all fully distributed and highly scalable. This way we can make sure none of these component will become a bottleneck for the system because of poor scalability.

Using SQS as a distributed queue enables us to use the pulling approach for load balancing and task distribution. Instead of putting an administrator component (often times centralized) to decide how to distribute the jobs between the worker nodes, the worker nodes decide when to pull the jobs and run them. The pulling mechanism has many benefits over the pushing. 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 and make decisions as well as distributing the jobs to among the workers. On pulling approach the only communication required is pulling the jobs. Using this approach can deliver good load balancing on worker nodes.

Using other third party cloud services, the CloudKon processing overhead is very low. 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.

Another advantage of using a distributed queue is decoupling different components of the system. Different components can operate independently with the SQS component in the middle to decouple different parts of the framework from each other. That makes our design compact, robust and easily extendable.

The scheduler can work in a cross-platform fashion with ability to serve on a heterogeneous environment that has systems with various types of nodes with different platforms and configurations. Using distributed queues also helps reducing the dependency between clients and the workers. The clients and workers can modify their pushing/pulling rate without having any effect on each other.

All of the advantages mentioned above rely on a distributed queue that could provide good performance in any scale. Amazon SQS is a highly scalable cloud 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 public cloud environment and runs on its resources optimally.

The system makes it easy for the users to run their jobs over the cloud resources in a distributed fashion just using a client front end without having to worry about setting up any cluster to run their jobs on.

## 3.1 Architecture Overview

This section explains about the system design of CloudKon. We have used a component based design on this project for two reasons:

- A component based design fits better in the cloud environment. It also helps designing the project in a loosely-coupled fashion.
- It will be easier to improve the implementation in the future.

Figure 1 shows the architecture of CloudKon. The client node works as a front end to the users to submit their tasks. The standard message format for SQS messages is String. SQS has a limit of 256 KB for the size of the messages. In order to send tasks via SQS we need to use an efficient serialization protocol with low processing overhead. We have considered using JSON and Google Protocol Buffer. After implementing our message with both serialization options, we chose Google Protocol Buffer because the size of the messages that was made with Google Protocol Buffer was 44% less than the size of JSON messages.

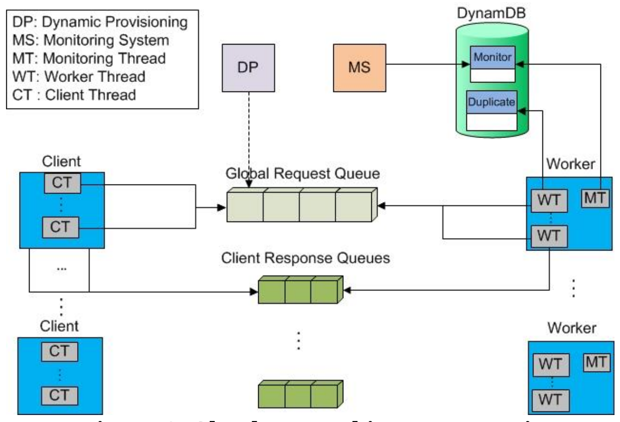

**Figure 1. CloudKon architecture overview**

CloudKon has many components that each has a separate independent responsibility. The components are:

- Global Request Queue
- Client Response Queues
- Client
- Worker
- Dynamic Provisioner
- Duplicate Task Controller
- Monitoring System

We have defined a message to use as a task container during the communication phase. Each message has a Task ID which is unique among tasks in each Client. It also includes different time stamps. Some of them include: Send time which is the time that the message will be sent by the client, and Receive time which is the time that a worker receives a task from the Global Request Queue. The Client ID is unique among all different Clients. We use an algorithm that provides unique global IDs so we don't have to worry about setting a policy to assign IDs to each client. Clients can start working independently without having a repeated Client ID. Worker ID will be left empty at the send time and is set by the worker that runs the task. The response queue address is specified for worker nodes to send back the results. Each Client has its own response queue. Finally the Body is the section that includes the command and its arguments. We have also left a reserved part for future uses of the project.

The Client code is multithreaded. That means it can submit multiple tasks to the SQS in parallel. After creating the task messages, Client threads submit multiple those in message batches that wrap up to 10 messages at each time of communicating with SQS. This way we can avoid the large overhead of communication up to 10 times.

Worker nodes on CloudKon have the ability to be launched and run independently without the need to register anywhere. This way we can have a scalable system with extreme number of worker nodes working independently. Worker code is also multithreaded and is able to receive multiple tasks in parallel. Each thread fetches up to 10 tasks message packages. Again, this feature is enabled to reduce the large communication overhead. After receiving a task, the worker thread has to verify that this is the first time that this task is being executed. After verifying that the task is being executed for the first time, the worker thread decomposes the message into the task. Then it fetches the task command and runs it. After finishing the execution, it puts the results into the message and sends it back to its corresponding client using the client response queue address field.

Soon after submitting the tasks, the client thread starts looking for the results in its particular response queue. The client does not stop until it gets back all of the results for the tasks.

## 3.2 Task Consistency on CloudKon

A limitation of SQS is that it does not guarantee delivering the messages exactly once. It guarantees delivering message at least once. That means there might be duplicate messages delivered to the workers. In order to be able to run many types of applications our system needs to guarantee the exactly once execution of the tasks.

In order to be able to verify the duplication we have used DynamoDB. After receiving a task, the worker thread has to verify that this is the first time that this task is being executed. DynamoDB provides a fast and simple key value store. Each time that a worker thread accesses this service it tries to add the unique identifier of a task which is a combination of the Task ID and the Client ID into the store. The operation succeeds if the message is not available in the store and is written for the first time. Otherwise the operation fails and the worker finds out that this was a duplicate message. This operation is an atomic operation. Using this technique we have minimized the number of communications between the worker and DynamoDB.

As we mentioned above, exactly once delivery is necessary for many type of applications such as scientific applications. But there are some applications that have more relaxed consistency requirements and can still function without this requirement. Our program has ability to disable this feature for these applications to reduce the latency and increase the total performance. We will study the overhead of this feature on the total performance of the system in the evaluation section.

## 3.3 Dynamic Provisioning

One of the main goals in the public 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.

We first considered using Amazon Cloud Watch for this purpose. 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. The problem with using Cloud Watch in our system is that the shortest period for updating the state of the SQS is 5 minutes, which is fine for industrial and Internet applications. But it is definitely not acceptable for our application. 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 global 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 set as program input by the user.

In order to use the resources efficiently, we have added a feature to the worker nodes. The worker node can deregister itself from the provisioner and terminate if two conditions hold. That only happens if the worker goes to the idle state for a while and also if the instance is getting close to its lease renewal. The instances in Amazon EC2 are charged on hourly basis and will get renewed every hour of the user don't shut them down. This mechanism helps our system scale down automatically without the need to get any request from a component. Using these mechanisms, the system is able to dynamically scale up and down.

## 3.4 Monitoring

Monitoring is an important feature for a job management/scheduling system. It can be useful for many purposes such as utilization monitoring and debugging. In order to provide monitoring on CloudKon, we have used DynamoDB. There is a monitoring thread running on each worker node to report specific details of the worker to the key value store periodically. Currently we are using the monitoring system to report the system utilization on worker nodes. The key value store in DynamoDB keeps track of all of the workers. The monitoring component reads the specific data it needs from the store in a real time fashion.

## 3.5 Communication Cost on CloudKon

The network latency between the instances in the public Cloud is relatively high compared to HPC systems. In order to maintain a service that can achieve a reasonable throughput and latency we need to minimize the communication between the different components of the system. Figure 2 shows the number of communications required to finish a complete cycle of running a task. There are 7 steps of communication to execute a task. At the first step, the Client sends the tasks to the global request queue in a single call. The worker then makes a call to the request queue and fetches the message at a single operation. After receiving a message, the worker makes a conditional write call to the DynamoDB system. After running the task, the Worker sends a message to the response queue. The execution cycle is completed by the Client when it gets the message from its response queue.

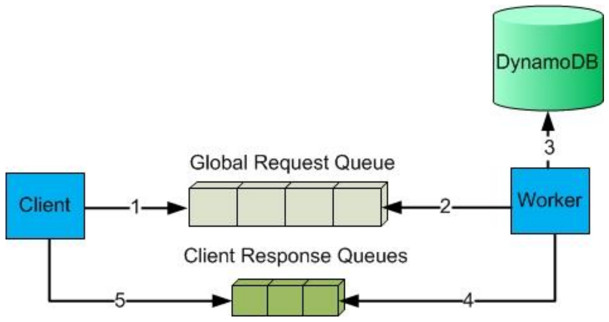

**Figure 2. Communication Cost**

In order to minimize the communication overhead, we also use message batching. This way we can send multiple tasks together. Figure 3 shows the number of messages we send on each communication between different components. The maximum message batch size in SQS is 256 KB or 10 messages. We have used message bundling on all of our communication steps except than one step. The Worker sends back the results to the response queue as soon as it runs the task. The reason for that is in order to send a batch of results to the response queue of a certain Client; the Worker needs wait until it runs a bunch of tasks from that certain client which is not desirable.

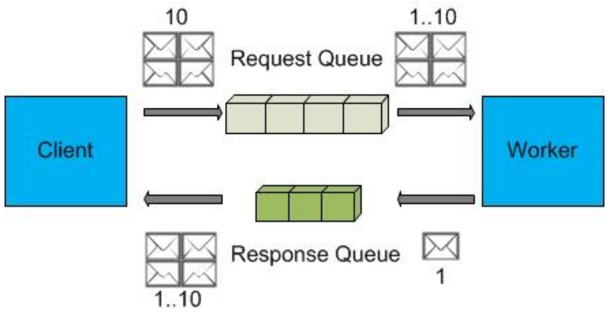

**Figure 3. Message Flow Diagram**

### 3.6 Security and Reliability on CloudKon

For the system security of CloudKon, we rely on the security of the SQS. SQS provides a highly secure and fast system using authentication mechanism. Only authorized users can access to the contents of the Queues. In order to keep the latency low, we don't add any encryption to the messages [18]. SQS provides reliability by storing the messages redundantly on multiple servers and in multiple data centers [18].

### 3.7 Implementation details of CloudKon

We have implemented all of the CloudKon components in Java. Our implementation is multithreaded in both Client and Worker component codes. 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.

We have used some open source libraries in our implementation. The libraries include:

- AWS Java SDK library, for communicating with different AWS services [34]
- Apache Commons library for Base 64 Encoding and decoding [35]

Making benefit of AWS service, our system has a short and simple code base. The code base of CloudKon is significantly shorter than other common task execution systems like Falkon or Sparrow. CloudKon code has about 1000 lines of code, while Falkon has 33000+ lines and Sparrow has 24000+ lines of code. This can show the potential benefits of the public cloud services. We can create a fairly complicated and scalable system by making benefit of already available system in the cloud.

## 4 PERFORMANCE EVALUATION

In this section we evaluate the performance of the CloudKon comparing it with other systems using different metrics. In all of our experiments, we have used m1.medium instances on Amazon EC2. We have run all of our experiments on us.east.1 datacenter of Amazon. We have used up to 64 nodes and 65 SQS queues in the experiments. In order to make the experiments efficient, client and worker nodes both run on each node. 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 with the help of some other programs like Parallel-SSH to run the experiments.

### 4.1 Throughput and latency on CloudKon

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. We have also compared the throughput and latency of CloudKon with Sparrow and Falkon.

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.

We have evaluated the throughput of CloudKon from 1 to 64 instances running 16000 to 1024000 tasks. The results show that CloudKon achieves almost linear speedup starting from 87 tasks per second on 1 instance to 5735 tasks per second on 64 instances. Therefore we predict that our solution scales at the same rate on larger scales.

Figure 4 compares the throughput of CloudKon with Sparrow and Falkon on different scales. We have used the same configuration on all of the systems running 2 client threads and 4 worker threads on each instance running 16000 tasks on each instance.

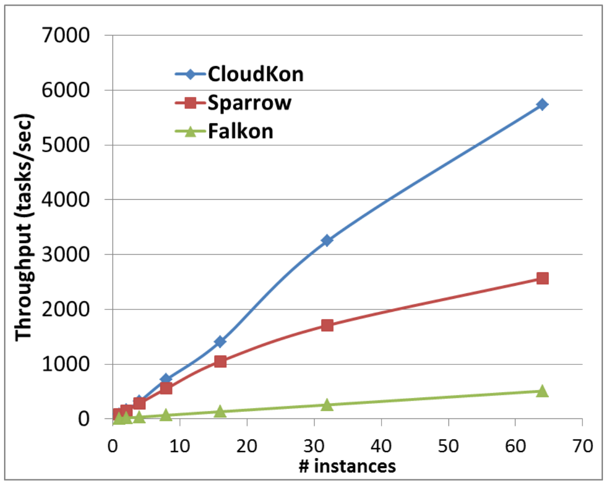

**Figure 4. Comparing the throughput of different job execution systems**

The results show that CloudKon was able to outperform the other two systems after the scale of 16 instances. CloudKon was able to achieve an almost linear speedup after the scale of 16 instances while the other two systems were not able to scale up perfectly. Falkon performs slower than the other two systems and cannot scale up linearly due to having a centralized architecture. One of the main reasons that CloudKon is outperforming the other two is being optimized for the public cloud environment.

Figure 5 compares the latency of CloudKon with Sparrow and Falkon on different scales. CloudKon has a lower latency comparing to the other two systems. The latency starts to increase after 32 instance scale on CloudKon. The reason for that is increasing the number of tasks on the request queue.

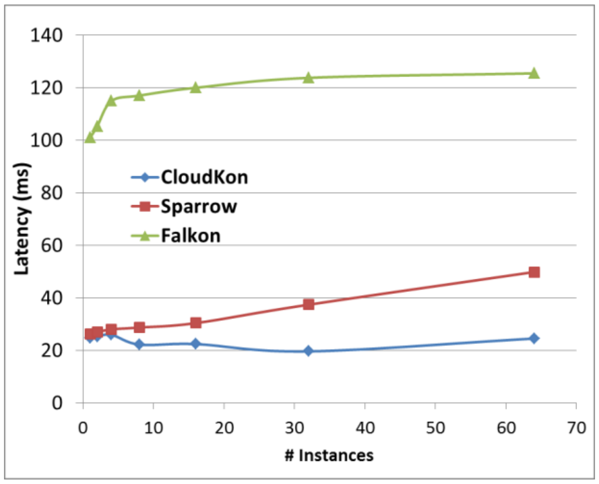

**Figure 5. Comparing the latency of different job execution systems**

## 4.2 The overhead of consistency on throughput and latency

As we have mentioned before, some applications have more relaxed requirements and can tolerate running the tasks more than once without generating any error. In this section we are going to evaluate the performance of the CloudKon when the duplicate task controller tool is disabled.

Figure 6 and 7 compare the throughput and latency of CloudKon when duplicate task controller is enabled/disabled. The results show that the throughput of the CloudKon when the duplicate is disabled is 1.5 times more on average. The throughput of the framework gets to 7991 tasks per task on 64 instances.

The latency of CloudKon decreases for 37% on average when the duplicate controller is off. The average latency of the framework is 15 ms comparing to 23.5 ms when the controller is enabled.

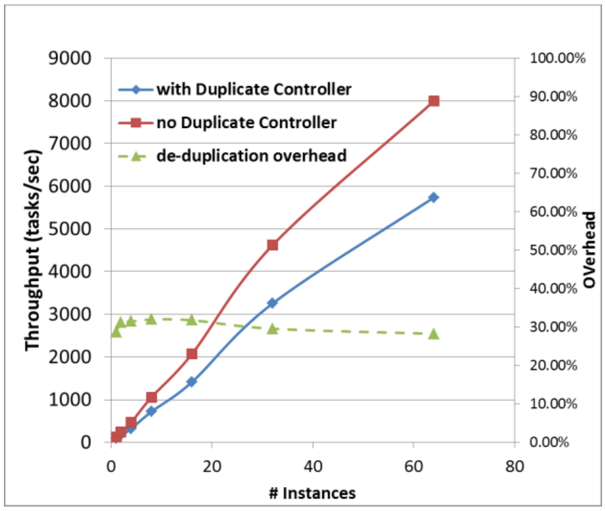

**Figure 6. System throughput when duplicate controller is enabled/disabled**

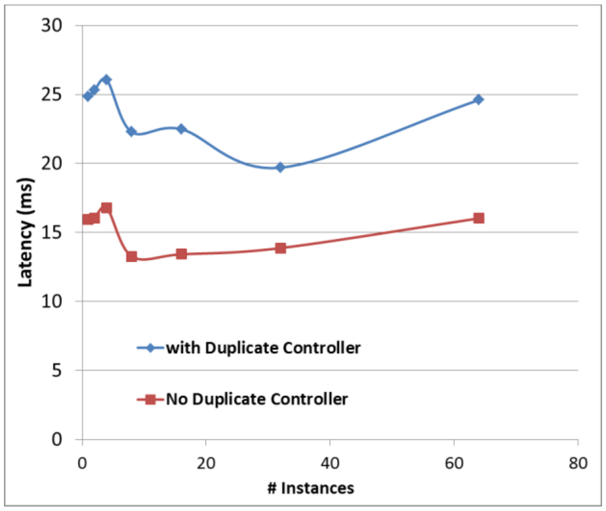

**Figure 7. System latency when duplicate controller is enabled/disabled**

## 4.3 Efficiency of CloudKon

Another important requirement for an execution system is to be efficient. The system should be able to run short tasks with less than a second task length efficiently.

Figure 8 compares the efficiency of CloudKon, Sparrow and the Falkon. The efficiency of our system is slightly better than Sparrow. The efficiency gets to 92% for tasks that take 1 second. This shows that CloudKon is a very light weight system that adds minimal overhead to the system.

## 4.4 The overhead of consistency on efficiency

In this section we evaluate effect of task consistency on the system efficiency. Figure 9 shows that the efficiency of the system without the controller is very good for shorter tasks with less than 1 second. The de-duplication effect decreases with the increase of the task length.

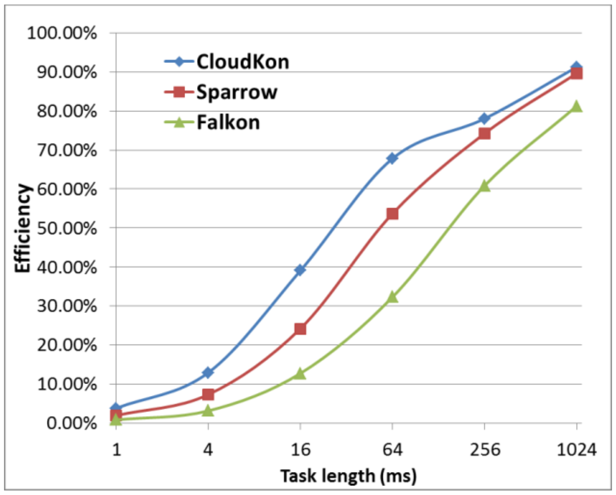

**Figure 8. System effiency of CloudKon compared to other task execution systems**

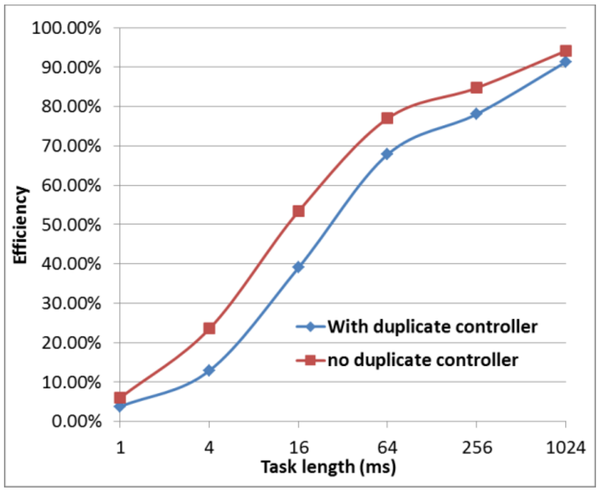

**Figure 9. System efficiency when duplicate controller is enabled/disabled**

## 5 CONCLUSION AND FUTURE WORK

Large scale distributed systems require efficient job scheduling system to achieve high throughput and system utilization. It is important for the scheduling system to provide high throughput and low latency on the larger scales and add minimal overhead to the workflow. CloudKon is a Cloud enabled distributed task execution framework that runs on Amazon AWS cloud. It uses SQS cloud service as a building block. SQS is a highly scalable 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 latency measurements show that CloudKon is a very lightweight system that adds minimal overhead to the workflow. 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 for CloudKon is to make the system 100% independent to be able to run it on different public and private clouds. In order to provide such system, we are going to implement a SQS like service with ability to provide high throughput content delivery at the larger access scales. With help from other systems such as ZHT distributed hash table [31] we will implement this queue in a way that can guarantee exactly once delivery. Another future direction of this work is to create a more tightly coupled implementation of CloudKon for HPC environments. We are also planning to evaluate the performance of the CloudKon on larger scales to find the limitations of the SQS service.

## 6 REFERENCES

[1] P. Kogge, et. al., “Exascale computing study: Technology challenges in achieving exascale systems,” 2008.

[2] 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.

[3] 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.

[4] J. Frey, T. Tannenbaum, I. Foster, M. Frey, S. Tuecke. “Condor-G: A Computation Management Agent for Multi-Institutional Grids,” Cluster Computing, 2002.

[5] B. Bode et. al. “The Portable Batch Scheduler and the Maui Scheduler on Linux Clusters,” Usenix, 4th Annual Linux Showcase & Conference, 2000.

[6] W. Gentzsch, et. al. “Sun Grid Engine: Towards Creating a Compute Power Grid,” 1st International Symposium on Cluster Computing and the Grid, 2001.

[7] C. Dumitrescu, I. Raicu, I. Foster. “Experiences in Running Workloads over Grid3”, The 4th International Conference on Grid and Cooperative Computing (GCC 2005)

[8] I. Raicu, et. al. “Toward Loosely Coupled Programming on Petascale Systems,” IEEE SC 2008.

[9] I. Raicu, et. al. “Falkon: A Fast and Light-weight tasK executiON Framework,” IEEE/ACM SC 2007.

[10] 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.

[11] 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.

[12] 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.

[13] Guohui Wang and T. S. Eugene Ng. The Impact of Virtualization on Network Performance of Amazon EC2 Data Center. In IEEE INFOCOM, 2010.

[14] 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.

[15] 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

[16] Amazon Elastic Compute Cloud (Amazon EC2), Amazon Web Services, [online] 2013, http://aws.amazon.com/ec2/

[17] Xen Hypervisor, xen.org, [online] 2013, http://www.xen.org/products/xenhyp.html

[18] Amazon SQS, [online] 2013, http://aws.amazon.com/sqs/

[19] LSF:http://platform.com/Products/TheLSFSuite/Batch, 2012.

[20] L. V. Kal´e et. al. “Comparing the performance of two dynamic load distribution methods,” In Proceedings of the 1988 International Conference on Parallel Processing, pages 8–11, August 1988.

[21] W. W. Shu and L. V. Kal´e, “A dynamic load balancing strategy for the Chare Kernel system,” In Proceedings of Supercomputing ’89, pages 389–398, November 1989.

[22] A. Sinha and L.V. Kal´e, “A load balancing strategy for prioritized execution of tasks,” In International Parallel Processing Symposium, pages 230–237, April 1993.

[23] M.H. Willebeek-LeMair, A.P. Reeves, “Strategies for dynamic load balancing on highly parallel computers,” In IEEE Transactions on Parallel and Distributed Systems, volume 4, September 1993

[24] G. Zhang, et. al, “Hierarchical Load Balancing for Charm++ Applications on Large Supercomputers,” In Proceedings of the 2010 39th International Conference on Parallel Processing Workshops, ICPPW 10, pages 436-444, Washington, DC, USA, 2010.

[25] OUSTERHOUT, K., WENDELL, P., ZAHARIA, M., AND STOICA, I. Sparrow: Scalable scheduling for sub-second parallel jobs. Tech. Rep. UCB/EECS-2013-29, EECS Department, University of California, Berkeley, Apr 2013.M.

[26] Frigo, et. al, “The implementation of the Cilk-5 multithreaded language,” In Proc. Conf. on Prog. Language Design and Implementation (PLDI), pages 212–223. ACM SIGPLAN, 1998.

[27] R. D. Blumofe, et. al. “Scheduling multithreaded computations by work stealing,” In Proc. 35th FOCS, pages 356–368, Nov. 1994.

[28] V. Kumar, et. al. “Scalable load balancing techniques for parallel computers,” J. Parallel Distrib. Comput., 22(1):60–79, 1994.

[29] J. Dinan et. al. “Scalable work stealing,” In Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis (SC '09), 2009.

[30] K.Wang, A. Rajendran, and I. Raicu. "MATRIX: Many-task computing execution fabric at exascale". 2013. Available from http: //datasys.cs.iit.edu/projects/MATRIX/index.html

[31] T. Li, et al., “ZHT: A light-weight reliable persistent dynamic scalable zero-hop distributed hash table,” in IEEE International Parallel & Distributed Processing Symposium, IEEE IPDPS ’13, 2013.

[32] Amazon DynamoDB (beta), Amazon Web Services, [online] 2013, http://aws.amazon.com/dynamodb

[33] P. Mell and T. Grance. NIST definition of cloud computing. National Institute of Standards and Technology. October 7, 2009.

[34] AWS SDK for Java, Amazon Web Services, [online] 2013, http://aws.amazon.com/sdkforjava

[35] Apache Commons codec, Apache Software Foundation, [online] 2013, http://commons.apache.org/proper/commons-codec

[36] MELNIK, S., GUBAREV, A., LONG, J. J., ROMER, G., SHIVAKUMAR, S., TOLTON, M., AND VASSILAKIS, T. Dremel: Interactive Analysis of Web-Scale Datasets. *Proc. VLDB Endow.* (2010).