Institute of Architecture of Application System
University of Stuttgart
Universitaetstrasse 38
70569 Stuttgart

Master Thesis Nr. 3142

Event-based Automated Management of Cloud Applications

Sams Ul Arefin

Course of Study: INFOTECH (MSc)

Examiner: Prof. Dr. Frank Leymann

Supervisor: Dipl.-Inf. Christoph Alexander Fehling

Commenced : 01 February, 2011

Completed : 02 August, 2011

CR-Classification: H.4.1, K.1, K.6.0, K.6.2, K.6.4, I.2.2, D.1.5

Contents

List of Figures

List of Listings

List of Algorithms

1. Introduction

Cloud computing has opened a new paradigm in the area of distributed computing in which users lease computing resources from large scale data centers operated by service providers. It delivers the large scale computation and data processing operation in a scalable and flexible manner. It provides the opportunity for small organizations and individuals to deploy applications by paying a minimal cost of actual resource usage [8]. Cloud resources such as *virtual machine* [52] can be provisioned on-demand on a *Pay-As-You-Go* basis [9]. Current resources provisioning approaches [9] [11] rely on monitoring the state of system resources such as CPU or memory utilization of virtual machines to determine the required size of the system. Most cloud service providers use machine virtualization to provide flexible and cost effective resource sharing. The cloud service provider is responsible to make the needed resources available on demand to the cloud users. Out of the three types (*Infrastructure-as-a-Service IaaS, Platform-as-a-Service PaaS* and *Software-as-a-Service SaaS*) of computing capacities as a service [12] in different abstraction levels, this thesis focused on Infrastructure-as-a-Service (*IaaS)* [5] only to provide computing resources as a service to customers. It introduces dynamic resource allocation mechanisms for infrastructure provisioning where there is a fixed time-limit as well as a resource budget for a particular task.

1.1 Motivation of the Thesis

1.1.1 Challenges

Though the dynamic provisioning of computing and storage resources in the Cloud is the major appeal to its users, it is quite challenging. The major challenge is to keep the resource budget to a minimum, while fulfilling the resource demand of an application. Current cloud service providers [51] follow *Pay-As-You-Go* [9] or the utility based pricing model [53] when they distribute the application workload among multiple instances where each instance work as a physical server. Usually an instance follows a time-based pricing model and charges user by hour as per the configuration of CPU cores, memory, and disk capacity. For example, *Amazon EC2* [20] users pay based on the type and number of the instances they consumed. *AWS Elastic Beanstalk* [52] provides application scalability by adding new instance without interrupting Clients' application to support increasing traffic growth. It allows a user to provision minimum of 1 and a maximum of 10,000 instances [1]. To determine the required number of instances is one of the major challenges here. Managing various multivariate uncertainties such as price, demand and availability also need to be considered. The Cloud user or the service provider needs to be accurate to reduce over provisioning cost as well as manage under provisioning cases. In addition, Cloud providers are also liable to meet *Service Level Agreements* (SLAs) [53] within deadline and budget constraint. All of these factors have made cloud dynamic resource provisioning a challenging job.

1.1.2 Goals

In this thesis, a method will be developed and investigated to optimize the workload management of Cloud-based applications. It offers cloud users to move workload to times when the required cloud resources are offered at better conditions and cloud providers to increase the utilization of the resources forming the offered cloud. In this setting, the cloud resources are considered to be accessed based on Service Level Agreements (SLA) subsuming a set of Service Level Objectives (SLO) that the cloud application needs to fulfill. SLA includes parameters like deployment deadline or preferred maximum pricing while negotiating between cloud users and providers. Similarly, the required response time can be used to form a SLO. Optimization will be performed through management of the workload and dynamic provisioning of cloud resources. This management function is triggered from a trigger management component that collects events originating from different application components. Triggers are sent to provision new resources as well as de-provision unused ones. An *Event* can be infrastructure based (e.g: CPU utilization of servers), platform based (e.g: the number of messages in a queue), application based (e.g: a certain application function is accessed) or user initiated (e.g: the user decides to suspend the complete application). Further, events can also be generated from the environment in case the price of the resource changes. Instead of handling each resource access as soon as possible it will be performed best fitting the required SLOs. To do so, cloud resources are monitored to determine so called Management Influencing Factors (MIF), such as provisioning time or request handling time. Based on this information the number of resources in the system can be optimized as well as the point in time at which resource accesses are granted.

This thesis will propose a optimize resource provisioning system which will be cost-efficient as well as highly resource utilized. For that, an event based workload management system is proposed which offers users to deploy their application into clouds based on SLA conditions. It will ensure maximum utilization of cloud resources by making delay of resource provisioning rather than following immediate request handling method.

1.2 Document Structure

The document is divided into the following chapters.

Chapter 2: Background and related work describes the resource planning in clouds as well as gives an analysis of existing scaling approaches for cloud and grid resources.

Chapter 3: Policy based scaling illustrates the conceptual aspect of the dynamic resource provisioning model.

Chapter 4: Implementation explains the system architecture which utilizes the information obtained from optimization processes and handles dynamic provisioning of cloud resources.

Chapter 6: Evaluation analyzes several scenarios to evaluate the performance of the policy based approach in comparison with others.

Chapter 7: Summary concludes the work by summarizing the results and gives an outlook on further improvements.

2. Background and Related Work

Virtualization [52] innovated since computing process starts. Computer has facilitated among the mass rather than few giant Companies in last decade. The key idea of cloud computing is its utility based approach which ensures on-demand provisioning and de-provisioning of resources by paying only for consumed resources. Cloud computing offers scalable resource provisioning which ensures overall cost reduction [22]. To do that, it requires an effective resource planning inside Cloud. This chapter illustrates the resource planning in the cloud and existing auto scaling mechanisms offered by different providers for the cloud environment.

2.1 Resource planning in Cloud

Dynamic provisioning is the key point while designing a system to put into the cloud. Besides, cloud based applications need to monitor constantly and manually adjust the needed cooling power. Usually, predictive statistical model enables release of over-provisioned resources whereas adaptive resource allocation minimizes consumer costs [27]. In recent years cloud scientist are trying to build a standard auto scaling approach. In order to optimize cloud resource provisioning cost, event based dynamic resource provisioning plan considered in this thesis.

- **Predictable demand on Time Slot**

We have few situations when workload on the application is incredibly high. Such as time between 7am and 7pm business hours as during this period the applications accessed by the employees of a company mostly. Another prediction would be during lunch and dinner time for an application that processes restaurant orders. So, we can predict the demand spike during the day and plan for scaling strategy. In this situation, AzureWatch allow the scheduling aspects into execution of a scaling rule [28].

- **React to Unpredictable Demand**

Cloud utilization metrics includes CPU utilization, amount of requests per second, number of concurrent users, amounts of bytes transferred and amount of memory used by applications. When utilization metrics shows high load, simply react by scaling up. For example, let's consider a site that may become incredibly popular suddenly and receive a large influx with visitors. Application that deals with disaster situations (earthquakes, tsunami, etc) could be an example here. In AzureWatch cloud user can configure scaling rules that aggregate metrics within particular time and send scale up trigger when the metrics is reach the threshold. For multiple metrics, need to find "common scaling unit" that would integrate all relevant metrics together into one number [27].

- **React to rate of change in unpredictable demand**

Considering the execution time of scale up and scale down events, we need to set point of reaction to initiate the request for scaling. AzureWatch's proposed a solution where an event can be set that interrogates average CPU utilization during last 20 minutes is 20% higher than average CPU utilization over the last hour and it already significant by being over 50%, then triggers of scale up initiated automatically [29].

- **Predictable demand from incomplete jobs**

Schedule-based demand prediction fits when the application workload can be determined by the amount of jobs waiting to be processed. Benefits of asynchronous job execution is achieved when high job processing is off-loaded to back-end servers and the amount of waiting time for to-be-processing jobs can serve as a metric. Scaling techniques of WindowsAzure use job scheduling mechanism is via Queues based on Azure Storage. Proposed AzureWatch provides the facility to create scaling rules based on the amount of messages waiting to be processed in a Queue [26].

- **On/Off Load**

In this scenario, workload usually occurs at certain periods. During other periods the application will be typically unused and kept be switched off. For example, financial batch processing jobs can be considered into this category.

- **Combine strategies**

While combining more than one of the mentioned scaling strategies, we need to consider known patterns for the applications behavior that would define the predictable bursting scenarios as well as take insurance policy to handle than unplanned bursts of demand [29]. Understanding the user demand and make scaling rules to work together is the key to success for auto scaling mechanism.

2.2 Analysis of existing scaling approaches for cloud and Grid resources

In Cloud computing world, several cloud approaches are offered by the companies like Amazon, Right Scale, MircoSoft, Scalar, Windows Azure, Rackspace and others. This section will briefly describe about those existing methodologies. Cloud platforms like Amazon EC2 or Windows Azure rely on various tools and services to provide dynamic auto scaling service, rather than automatic adjustment of computing power dedicated to applications running on their platforms. Amazon uses auto-scaling method via a service CloudWatch and third party vendor's (RightScale) tool for the applications running on their platform. On the other hand Windows Azure offers third party vendors AzureWatch to provide auto scaling and monitoring facility [22].

2.2.1 Amazon Auto Scaling

Auto Scaling is one of the web services from Amazon which was designed to start or terminate EC2 instances automatically based on user-defined policies, schedules and health checks. Auto Scaling is used to maintain the workload of *Amazon EC2 Instances* [20]. An *EC2 Instance* is a Virtual machine that provides a predictable amount of dedicated compute capacity and is charged per instance-hour consumed. Auto Scaling launches an additional instance whenever CPU usage exceeds 90 percent for last 10 minutes and terminates when CPU usage is half of especially during weekend. Auto scaling groups work across multiple physical locations to host EC2 instances called availability Zones. Auto Scaling automatically redistributes the cloud based applications into different *Availability Zones* [30]. Cloud Provider can set auto Scaling group so that user requests are distributed over a group of EC2 instances. Auto Scaling also supports AWS Elastic *Load Balancing* [56]. Adding an Elastic load Balancer is also possible to Auto Scaling group to measure request latency to scale the applications workload.

- **Auto Scaling Group**

EC2 instances [20] are categorized into Auto Scaling groups. Each group is defined with a minimum and maximum number of EC2 instances. The Auto Scaling service launches more instances for the Auto Scaling group to handle the increasing traffic and shut down the instances when demand decreases to ensure optimize usage of computing resources.

As shown in figure 2.2.1., internet traffic is routed from the public URL to an Auto Scaling group named "webtier". The Auto Scaling group triggers to increase or decrease the size of

group based on the average CPU utilization of the group. A trigger is a signal that lets the system when to increase or decrease the number of instances.

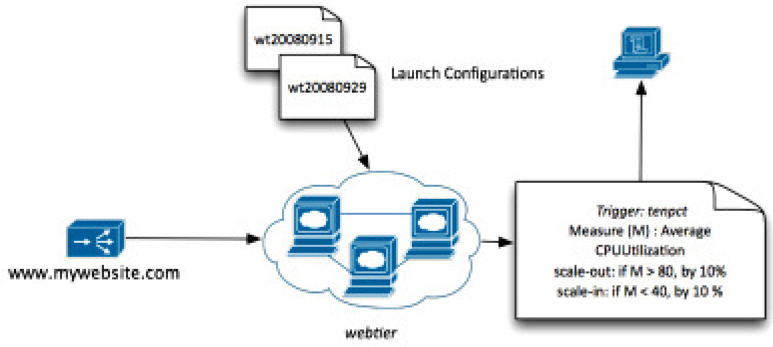

wt20080915
wt20080929
Launch Configurations
www.mywebsite.com
webtier
Trigger: tenpct
Measure [M] : Average CPUUtilization
scale-out: if M > 80, by 10%
scale-in: if M < 40, by 10 %

Figure 2.2.1: Auto scaling mechanism [31]

In Amazon auto scaling, user can set a trigger to activate on any of the metrics published to Amazon CloudWatch, such as CPU Utilization. When activated, the trigger launches a long-running process called a Scaling Activity. When a trigger fires, Auto Scaling uses a launch configuration to create a new instance [31].

- **Health Check**

A health check is the process to monitor the health status of each instance in an auto Scaling group. In case of having degraded performance, auto Scaling terminates that instance and launches another fresh one as replacement.

- **Launch Configuration**

A launch configuration captures the parameters necessary to create new EC2 Instances. An Auto Scaling group can have only one launch configuration at a time which is modifiable but has no changes on existing instances. A single AWS account has maximum 100 launch configurations. When Auto Scaling needs to scale down, it first terminates instances that have an older launch configuration [30].

- **Trigger**

A trigger combines Amazon CloudWatch alarm [30] and Auto Scaling policy together and inform what will occur when the alarm threshold is crossed. We need one trigger for scaling up and another for scaling down individually. For example, to scale up the instance when the CPU usage increases to 80 percent, we need to configure a CloudWatch alarm and Auto Scaling policy. The alarm sends a message to auto scaling as soon as the CPU usage has reached to 80 percent of its usage. When the CPU usage decreases to 40 percent, second trigger is sent to scale down.

- ## Policy

A set of instructions that instructs how to respond when *CloudWatch* [28] alarm messages are sent. User can configure a CloudWatch alarm to send a message to the Auto Scaling process whenever a specific metric has reached into a triggering value. When the alarm sends the message, Auto Scaling executes the associated policy on an auto Scaling group to scale the group up or down [29].

- ## Instance Distribution and Balance across Multiple Zones

To provide high scalability and reliability, Amazon data center facilities are located in several different physical locations which named as "*Regions and Availability Zones*". Amazon has four Regions: the US-East (Northern Virginia) Region (also known as the US Standard Region), the US-West (Northern California) Region, the Asia Pacific (Singapore) Region, and the EU (Ireland) Region. Availability Zones are unique locations within a region that are not affected by the failure in other Availability Zones and deliver inexpensive, low-latency network connectivity among other zones in same region.

Auto Scaling distributes instances uniformly between the Availability Zones which are connected to the user auto Scaling group. Auto Scaling attempts to launch new instances in the *Availability Zone* with the fewest instances. When failure occurred in one zone, Auto Scaling will attempt to launch in other zones until it succeeds [30].

- ## Types of Scaling

Amazon auto Scaling facilitate following three types of scaling Manual Scaling mechanism.

Manual scaling

In manual scaling, we need to call an API to use the Auto Scaling command line interface (CLI) to launch or terminate an Amazon EC2 instance. Cloud users need to specify the amount of capacity. This Scaling manages the process of creating or destroying instances, including all the parameters to the Amazon EC2 run Instance call [29].

Scaling by Schedule

When the demand of the instance is predicted, the scaling mechanism increases or decreases instance into the group in schedule basis. Scaling by schedule means that scaling actions are performed automatically as a function of time and date. To create a time-based scaling plan, user needs to specify the time at which the plan needs to take effect as well as tells the new, minimum and maximum instance size that require at that time. At that specified time, Auto Scaling updates the group to set the new values according to the scaling plan.

Scaling by Policy

Scaling by policy allow the cloud user to define the parameters that will be used in Auto Scaling process. A policy can be enlarged to the group whenever the average CPU utilization rate is higher than 90% for last 15 minutes. Users need to have two policies, one for scaling up and another for scaling down for each event that needs to monitor. Policy1 scales up when the network bandwidth reaches a certain level by firing up a certain number of the instances to help the request traffic. Again, policy2 scales down by a certain number when the network bandwidth level goes back down.

For example, we create a policy that allows developer to change the capacity of an auto scaling group and gives access to the *SetDesiredCapacity* [13] action.

```
{   "Statement":[{

"Effect":"Allow",
"Action":"autoscaling:SetDesiredCapacity",
"Resource":"*"
} ] }
```

• Auto Scaling Configuration

While configuring auto scaling mechanism, one the following services need to consider.

Maintain current instance levels

Health status of the instance are monitored in auto scaling system .When Auto Scaling finds an unhealthy instance, it terminates that and starts a new one. Auto Scaling considered the notification received from Amazon EC2, Elastic Load Balancing [56] and s*etHealthStatus* [31] API before declaring an instance as unhealthy.

Create more instances

A scaling policy indicates auto scaling to perform a scaling action when the metric value crossed the threshold [30]. For example, a policy can be 10 percent increase of current instances when the CPU utilization of the scaling group reaches 80 percent.

Delete current instances

Auto scaling performs a scaling down action by deleting instance when the metric value is too low [31]. For example, a policy can be 50 percent decrease of current instances when the CPU utilization of the scaling group goes down by 50 percent during last 30 minutes.

• Amazon CloudWatch

To monitor auto scaling group, Amazon use monitoring tool "CloudWatch". Amazon CloudWatch provides monitoring for AWS cloud resources, such as resource utilization, operational performance and overall demand patterns. It also includes metrics such as CPU utilization, disk reads and writes and network traffic [60].

Basic Monitoring

To create a new Auto Scaling group with basic monitoring, we have to create a lunch configuration that has the ***InstanceMonitoring.Enabled*** [35] flag which needs to set false.

Detial Monitoring

To enable the detailed instance monitoring for a new Auto Scaling group, user need to create a launch configuration. Each launch configuration contains a flag named *InstanceMonitoring. Enabled[35].* The default value of this flag is true.

• Challenges in Auto scaling & Proposed Solution

Though Amazon Auto scaling provides the policy based scaling, it is only useful when user can define the way to scale in response to changing conditions. However, user is not aware when those conditions will change. In this aspect, auto scaling need to adopt fault resilient dynamic load balancing [29].

As shown in figure 2.2.2, group membership scheme has considered for this load balancing technique. For fault resilience, single point of failure is avoided; the load balancers need to be replicated. It can handle multiple application groups where the application member can join or leave their own groups. So, the load balancers need to be part of the application groups as shown in figure 2.2.3. Here, the load balancers will be acknowledged about membership changes in the load balancer group as well as the application groups. However, the application group members are not be concern about membership changes in the load balancer group [29].

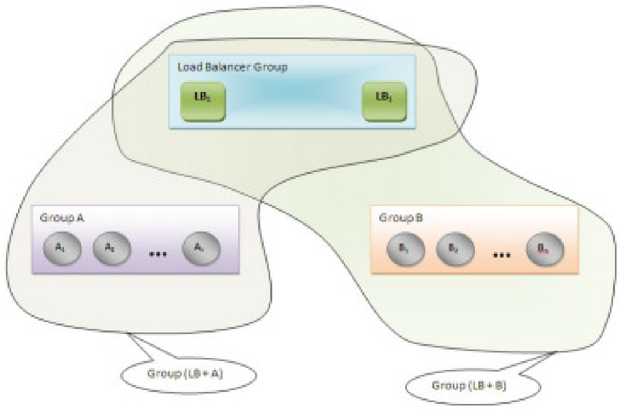


Figure 2.2.2: Load Balancer and application group [29]

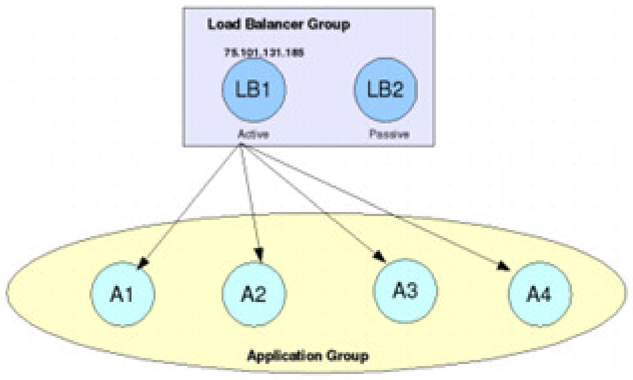


Figure 2.2.3 Active –passive load Balancer with elastic IP [29]

2.2.2 Amazon Beanstalk

AWS Elastic Beanstalk provides quick deployment functionality and manages the application in the AWS cloud. Cloud user needs to upload application and AWS Beanstalk automatically handles the deployment details of capacity provisioning, load balancing, auto-scaling, and application health monitoring. It also provides users opportunity to access the underlying resources at any time and facilitate other AWS services such as Amazon EC2, Amazon S3, Elastic Load Balancing, and Auto-Scaling [3].

AWS Elastic Beanstalk runs on the Amazon Linux AMI in order to deliver a stable, secure, and high performance execution environment for EC2 cloud computing. It consist a container built for Java developers using Apache Tomcat software stack. It takes few minutes to allocate the

AWS resources based on size of the deployable code, the number of application server user want to deploy. However, deployment of the new application version to the existing resources takes less time depending on the size of the version. User application can be scaled automatically tens or even hundreds of times on the basis threshold factors such as CPU utilization or network bandwidth. Thresholds can be configured for a particular application using AWS Elastic Beanstalk's management console. Usually, one AWS user account allows launching maximum of 20 EC2 instances and creating up to 10 Elastic Load Balancers [32].

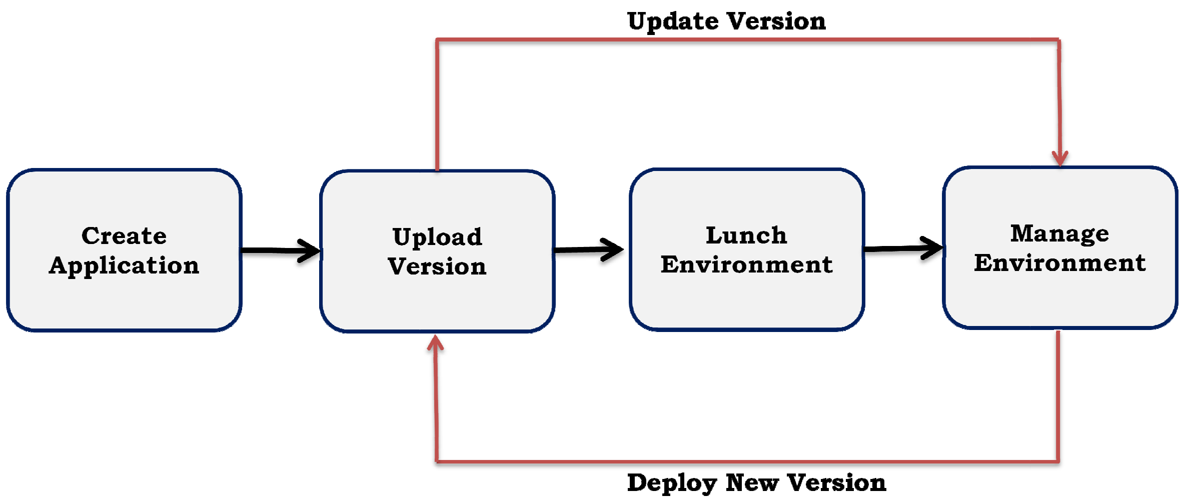


Figure 2.2.4 Concept of Elastic Beanstalk [32].

As depicted in figure 2.2.4, user needs to upload an application to Elastic Beanstalk with the certain information such as type or size of the application. AWS Elastic Beanstalk automatically creates and configures the AWS resources needed to run codes and no manual configuration for server capacity, load balancing and scaling is required for the application [33].

2.2.3 Scalr

Scalr is an open-sourced framework for managing the massive serving power of EC2 cloud. Scalr makes Amazon EC2 more exciting to the developers' community for its redundant, self-curing and self-scaling network [34].

- **Auto Scaling Verses Scalr**

EC2 auto scaling and Scalr have major difference in scaling mechanism. For example, let's consider a provider who has 3 web servers where each can handle maximum of 1000 concurrent users with adequate performance. At 5 am morning, traffic is low and it only has 1500 concurrent users. Since this is under the threshold, so scaling down event is triggered. In EC2 auto scaling, one of the servers will be terminated to optimize the resource usage. At this point, 500 users that were on the server lost their connections closed and be logged out if it used file based sessions with PHP and similar type. Similarly, for the web sockets and persistent connections would be a problem [33].

On the other hand, Scalr handles downscaling in different approach. Before terminating a server, the *onBeforeHostTerminate* [58] event is triggered which allows to perform maintenance actions to prepare the server for being terminated. Scalr uses the web server safe shutdown method that ensure no new connections are made, rather waits until all the existing connections are closed to terminate the server [35].

• Architecture

The Scalr framework is a series of server images, called Amazon Machine Images [57]. An AMI is a special type of pre-configured operating system and virtual application software which is used to create a Virtual Machine [52] within the Amazon Elastic Compute Cloud (EC2). Every basic website requires an app server, a load balancer, and a database server. The AMI consists of a management suite that monitors the load and operating status of different servers in the cloud. Scalr can increase or decrease capacity as per demand changes, as well as detects and rebuilds unhealthy instances. User has the administration privilege to control Amazon EC2 through the Scalr platform.

DNSManager

Scalr uses DNS zone to perform its scaling method. It uses four network-independent *nameservers* [61] ns1.scalr.net to ns4.scalr.net. NS1 and NS2 are based off the cloud at a datacenter in the US called The Planet, NS3 is on EC2 on the East Coast of the US and NS4 is equally on EC2 in Europe (Ireland). Scalr suggests user to reserve *nameserver* as per incoming traffic location that means the order NS1, NS2, NS3 and NS4 for the user placed in US and Asia [33].

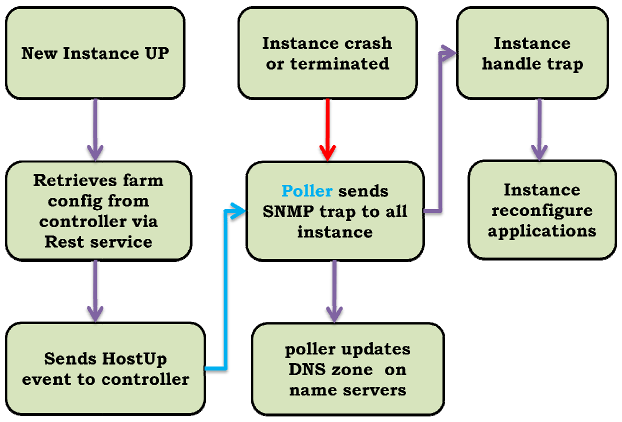

New Instance UP
Instance crash or terminated
Instance handle trap
Retrieves farm config from controller via Rest service
Poller sends SNMP trap to all instance
Instance reconfigure applications
Sends HostUp event to controller
poller updates DNS zone on name servers

Figure 2.2.5: Scalr Architecture [34]

Poller

As seen in figure 2.2.5 Scalr polls the Cloud provider to retrieve its status for every instance. Polling is done with one minute interval. Scalr updates the database according to the Cloud provider report, such as if the Cloud provider reports the server as running, but in Scalr's database it is marked as pending, Scalr updates the database as 'running'. Similarly, if the user terminates an instance, Scalr's database will be updated as pending terminate' state. It was marked for termination in the Scalr database, the API call to terminate the instance has been made and Scalr waits for the Cloud provider report as terminated before changing status terminated. For an instance is in the 'pending launch' state, this is just reverse [34].

Scaler

Scaler is a component of Scalr which compares the number of instances running for a particular role to the ***minimum_instances*** [33] and ***maximum_instances*** [33] values set in the farm configuration. If the number is below, Scalr launches additional instances to meet the minimum and terminates instances when the number is over the threshold. After this, the Scaler will check each Scalr algorithm and repeat the mentioned comparison with the thresholds and boundaries set by the user. It will also check the date and time, and compare with Schedule-based scaling [34].

2.2.4 Scaling in Windows Azure

Windows Azure platform proposed compute and storage capabilities required by cloud-based applications and other constituent services such as service bus and access control [36]. Windows Azure also provides robust dynamic scaling capabilities through a custom-coding approach. Third-party tools such as AzureWatch [38] monitors and dynamically scales Azure application. It is customizable rule-based engine that aggregates the performance counters, queue size and notify users to scale instances up or down based on configuration. In Windows Azure, auto scaling is achieved by changing the instance count in the service configuration. Increasing the instance count will cause Windows Azure to start new instance; decreasing the instance count will in turn cause it to shut instances down [38]. In this section, we will find the proposed dynamic scaling mechanism for Window Azure.

- **Workload Management in Azure**

Systems which have highly unpredictable load within short period need to have additional capacity since to start up Windows Azure instance cloud users need to wait certain interval. Since the instances are charged by hourly usage, so the load variance (up and then down) that occurs in every hour need to manage efficiently by running required instances to handle the hourly peak load.

Windows Azure supports the concept of Small, Medium, Large and Extra Large instances. It may contain one, two, four or 8 processors equally increased amounts of RAM and local storage. While some applications may be benefited from the usage of large sized Azure instance to improve the throughput, others are missing the elastic scalability due to the change in VM size which requires redeploy the service engine [36].

• Rules based Scaling

Though both EC2 and Azure provide auto scaling feature, but the major difference is the architecture of two systems. EC2 provides a backbone and framework for auto scaling, whereas Azure provides an API that can be extended. Some of the third-party providers are delivering tools for Azure auto scaling.

A set of rules decide when to scale and by how will do. Windows Azure uses two categories of rule for scaling applications. One of that is a rule which define to add or remove the capacity from the system in particular time such as run 20 instances between 13.00-18.00 hrs and only 2 instances in other time every day. Another form is a rule based on a response to the metrics and negotiation between service provider & user [37].

• Load Metrics

A metric based rule considers monitoring aspects of the system which may change over time and take decisions to scale up or down. In this section, we will find some of the metrics that are considered in Windows Azure scaling [37].

Primary Metrics

Primary metrics define the measurement of the work that are currently running into the system, such as the number of requests per second, the number of queue messages being processed per second and so on. Due of the highly stochastic nature of the most load profiles, it is necessary to use some algorithm to make this metric smooth. A moving average would be sufficient one [36].

Secondary Metrics

Secondary metrics measure the result of the load that already applied to the system such as CPU utilization, length of the queue, response time and so on. Few of these metrics are valuable in time value; others will require smoothing [36].

Derivative Metrics

Derivative metrics are derived from other metrics. Rate of change in the queue length, the rate of change in requests per second are the examples of this type of metrics. Derivative metrics model the acceleration or deceleration of changes in load. It ensures the optimize capacity, rather than adding far more capacity than required.

Accretive Metrics

These metrics track things that occur over time. It could include the total number of users of an application, the spent time on Windows Azure Platform services for the given period, etc.

• Evaluating Business Rules

Azure evaluates business rules to determine a proposed action. Few of these rules are complex and others are simple enough to code. Such as a rule that will take an action when a metric crosses a given threshold is pretty straight forward. Followings are few of the examples of rules.

- Minimum no of instance is 10 and maximum is 50.
- If the average response time during last 30 seconds exceeds 500ms then add additional instance.
- If the average response time over 30 seconds drops under 100ms then remove instance.
- Apply these rules until Monthly budget exceeds $4000 and inform Cloud user whether additional expense need to meet deadline.

While adding and removing instances it is necessary to be aware about change that was requested and need to consider when that change had completed. Azure applications need to ensure that rules are cognizant of the time delay while adding increasing capacity [38].

• Azure Scaling Example

This section describes a sample scaling engine for *Windows Azure Compute instances* [48] which considers two business rules. Assume that the business can determine minimum and maximum number of instances to be running at specific time period where weekdays and weekends have different thresholds. It is expected that number of target application users during weekend are lower than weekdays. However, it may differ during particular occasion or reason such as application which broadcast the live score of WorldCup football tournament in weekend. Subject to the threshold that is set for the day of a week, the application takes the scaling decisions around the collection of metrics. It monitors the current length of the queue and the current number of requests per second to the web service that delivers up the application. If the length of the queue grows quickly during a specific time period, a new worker instance is started. [37].

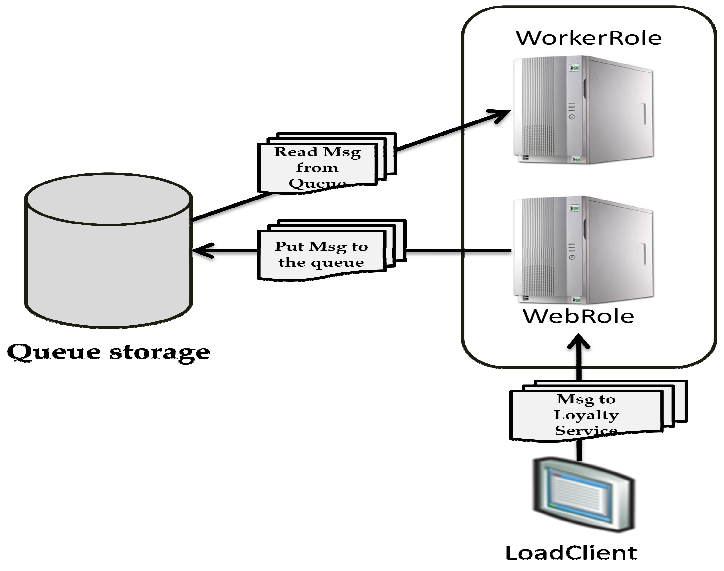


Figure 2.2.6: Load management in Windows Azure [37]

As shown in figure 2.2.6, ***Loyalty Management*** service [37] which contains a web role and a worker role. The ***web role*** adds items to a queue. The ***worker role*** picks items from the queue and processes them. This application scales the instance count for the two roles according to two parameters: amount of messages in the queue and the day of the week. Queue storage contains the queue list. ***Load Client*** is a client application that calls the web service hosted by the Loyalty Management web role. It can call the *Loyalty Management* service maximum 4 times per second. The *Load Client* is used to generate different levels of load on the application.

As shown in figure 2.2.7 ***Scaling Engine*** is the core part which is responsible for enforcing the scaling rules. The scaling engine can be placed in the cloud or on-premise. In this example, it is built as a console application running on-premise. It collects metrics such as the queue length and ***Performance counter*** (the number of web requests per second) from the Loyalty Management application. Using these metrics, it determines when to scale up or down the number of Azure instances. If scaling is needed, it calls the Azure Service Management API to start instance. ***Table storage*** stores the counts of performance counter and the queue length in the table. Scaling engine put the queue length one of the tables in table storage. It reads the performance counter from the table storage [37].

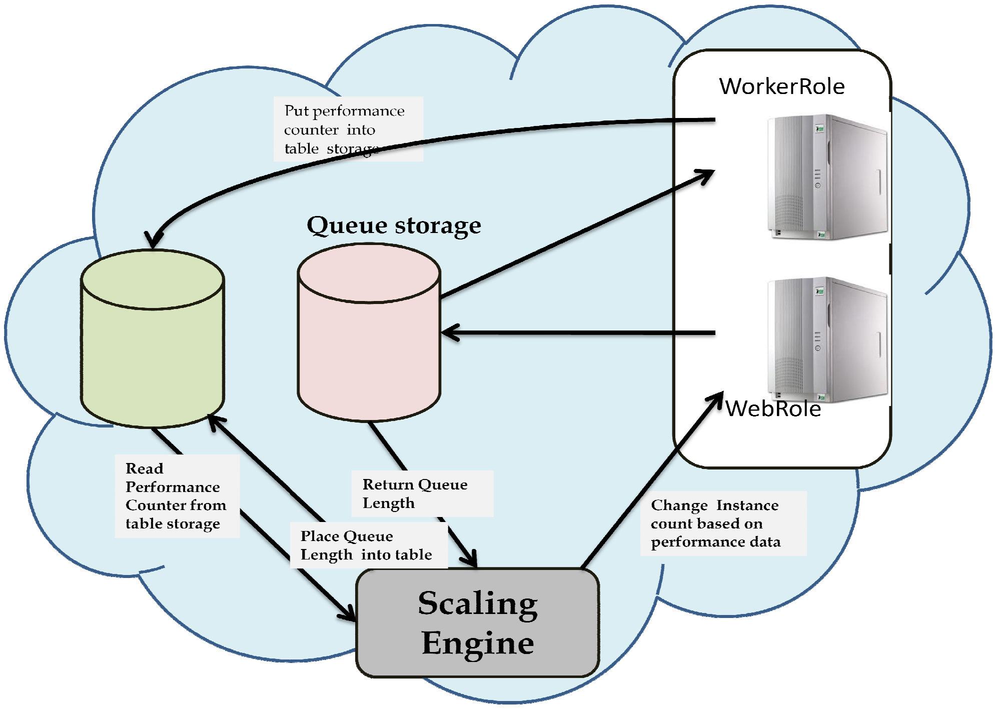

WorkerRole
Put performance counter into table storage
Queue storage
WebRole
Read Performance Counter from table storage
Return Queue Length
Place Queue Length into table
Change Instance count based on performance data
Scaling Engine

Figure 2.2.7: Architecture of Scaling Engine in Windows Azure [37]

- **AzureWatch**

AzureWatch works on Microsoft cloud platform and adds dynamic scaling capabilities to the applications running into it. It was developed by a company named "Paraleap Technologies". AzureWatch dynamically adjusts the number of compute instances dedicated to Azure application according to real time demand. User-defined rules define when to scale up or down to provide optimize computing power during job processing. AzureWatch plays the role in Windows Azure similar to the CloudWatch in EC2 [39].

AzureWatch monitors, aggregates and analyzes performance metrics from Azure applications and compare the metrics against user-defined rules in every minute once. When a rule satisfies the condition, scaling action occurs. Storage, aggregation and rule evaluation occurs on scalable *AzureWatch* servers running in the cloud. The process of monitoring metrics is done either from host machine or its servers. It includes windows-based Control Panel utility to setup and configure different rules which has the dashboard with historical reports and charts.

AzureWatch scale-up or scale-down Azure instances based on Real-time demand using latest values of performance counters, historical demands, rate of increase or decrease demand, queue size, reaction time to deploy instance and date-time. It also sends email alerts as soon as user-defined conditions are met. Safety mechanisms like built-in limits within predefined range and built-in throttle controls are also included with it [39].

2.2.5 RightScale

The RightScale Cloud Management Platform provides scalable and cost-effective IT infrastructure on demand. It reduces the complexity of cloud computing by allowing organizations to deploy the business-critical applications [40]. Customers can easily deploy, manage and dynamically scale complex, multi-cloud applications on this automated platform. RightScale allows setting up and configuring the necessary trigger points, called *Alerts* that automatically react to various monitored conditions when thresholds are exceeded. It provides dynamic auto scaling which scales application servers to respond on demand and configures the threshold that needs to be exceeded to start scaling and how fast the scaling should be achieved. The *ServerTemplates* [42] and the RightScale API are used to manage functional groups of servers, facilitate change and ensure reliability in the production environments. The elegance and power of the *ServerTemplates* results massive time savings [41].

- **Scalable Website Deployment**

RightScale provides scalable website deployment facility that runs a full customer-facing website on cloud infrastructures with scalability and reliability. As shown in figure 2.2.8 it includes with two load balancers, multiple application servers and replicated MySQL database servers for the recovery and rolling backups to the cloud storage. Based on the user defined rules, it scales up as demand grows and scales down as demand decreases. User decides when and how fast to scale. The clone capability creates a test environment in which users can assess performance under load and test their application in the cloud. It ensures load-balanced at front-ends and provides graphical monitoring and alerting.

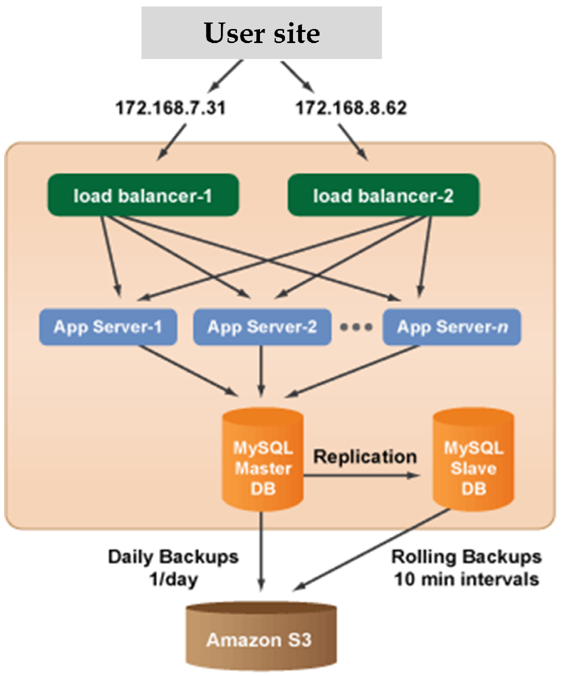


Figure 2.2.8: Scalable website deployment [44]

• Development and Test in RightScale

The RightScale provides development and test Solution pack to launch resources in the cloud with the specific configurations. Each user can easily launch and provision the specific configuration that needs in every phase of the development cycle. Developers can launch an all-in-one environment that runs the entire system (OS, app server + database) in a single machine. Testers can twist up *3-tier architectures* [63] where tier 1 is the *Load balancer*, tier 2 is the *Application server* [63] layer and tier 3 runs a single *Database server* [63]. Software Architects can get the similar to production environments to test application availability and reliability using redundant load balancers, a scalable app server layer and master and slave database servers. The Cloud Management Platform includes two user interfaces. A Self-Service Portal designed for developers and testers to launch servers in the cloud. The Management Dashboard is designed for systems administrators to customize the pre-configured environments delivered with the solution [43]. For example, an Internet content sharing network achieved 70% savings by setting up standalone deployments for software developers in the RightScale's cloud. Whenever needed, developers launch pre-configured environments and then decommission them when it is completed. Shutting down cloud environments on weekends and overnight provides significant savings. For that, developers must be able to preserve the environment and easily re-launch it. That's where RightScale's management platform adds momentous value to the customer [44].

• Grid Processing

In RightScale's *Grid Computing* Solution Pack user gets a complete, scalable grid application environment in the cloud. It includes a preconfigured framework to deploy grid processes that are automated, error resilient and fully auditable. It is designed to leverage Amazon Web Services specifically *Amazon EC2* [21], *Simple Queue Service* [17] and *Amazon S3* [45] capabilities that processes large numbers of jobs in a scalable and cost efficient manner. *Grid processing* serves compute-intensive applications where algorithms and data require massive amounts of computing power. The number of servers set up for grid processing is often limited due to the available tools and complex data processing requirements that result low capacity utilization over time. To solve this issue, *Grid processing* offers super high computing power with an acquisition model that allows businesses to pay just for what is used [44].

As shown in figure 2.2.9 Grid processing requires three parameters from users. User needs to define Minimum and maximum worker array sizes which will be used to scale up and down. It also requires some basic policies from user for scaling up and down. In addition it is necessary to mention the locations for the input data and where results should be stored.

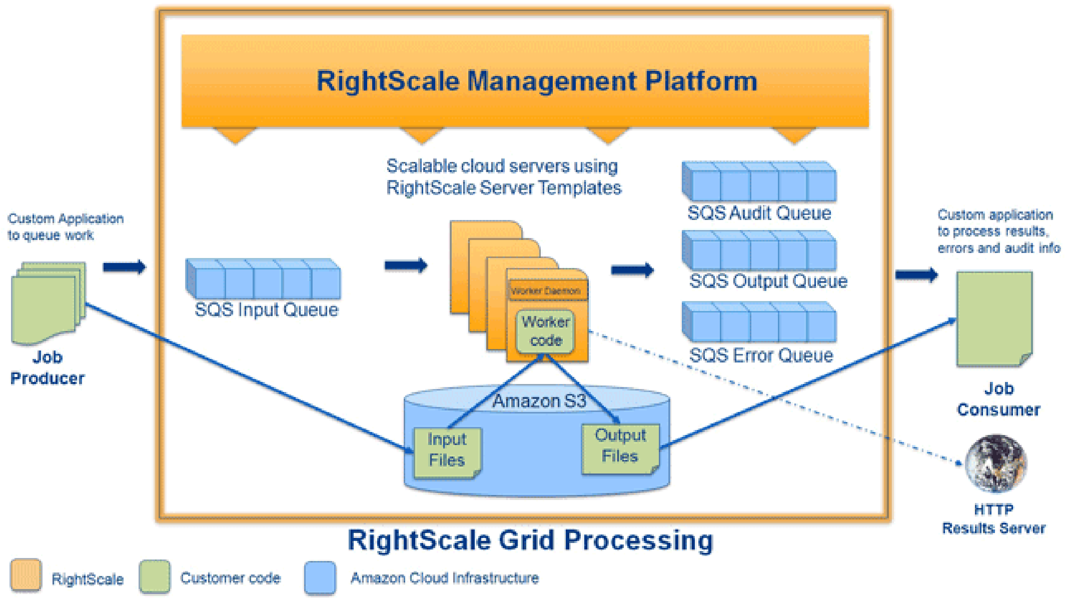

RightScale Management Platform
Scalable cloud servers using RightScale Server Templates
SQS Audit Queue
SQS Output Queue
SQS Error Queue
Custom Application to queue work
Job Producer
SQS Input Queue
Worker Daemon
Worker code
Amazon S3
Input Files
Output Files
Custom application to process results, errors and audit info
Job Consumer
HTTP Results Server
RightScale Grid Processing
RightScale
Customer code
Amazon Cloud Infrastructure

Figure 2.2.9: RightScale Grid processing [47]

RightScale provides batch computing solution that allows customers to utilize the on demand, virtually unlimited cloud resources for grid processing. The benefits of using the Grid Solution comes out in two different forms

- Cloud Resources can be accessed for a limited time and then turned off. Users pay only for what they actually used.
- In time critical projects, user can access massively parallel resources beyond their capacity or budget in most of the internal data centers [47]. As a result, they get their projects completed in exceptional timeframe.

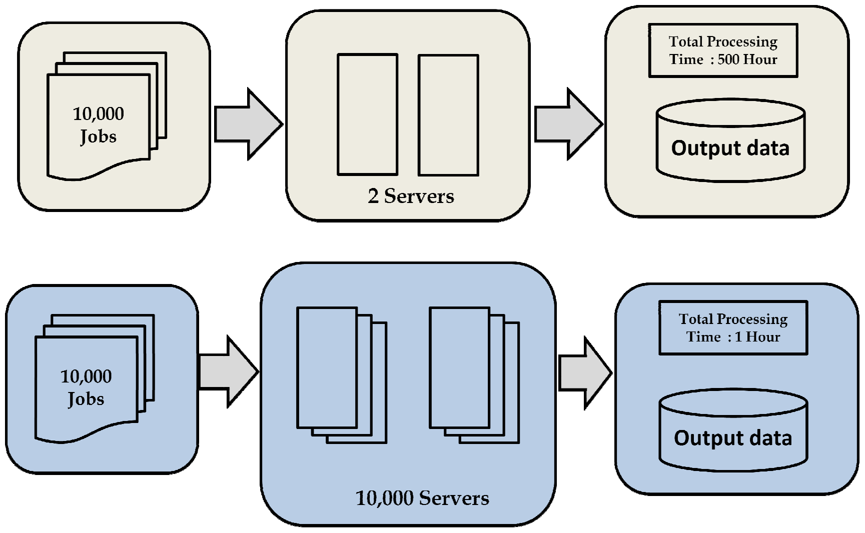


Figure 2.2.10 Job processing in Local data Canter and RightScale platform [44]

Figure 2.2.10 illustrates the savings between the virtually infinite grid capacity in RightScale's infrastructure and running a finite number of dedicated machines in the internal data center. As seen in figure 2.2.10, the task is to run 10,000 jobs with a single server instance (which process 10 jobs per hour) require 1,000 hours of compute time. Since pricing is based on usage, running the 1,000-hour project in the cloud costs same as single serve. However, when it runs on two cloud servers for 500 hours (three weeks) or for one hour on 1,000 cloud servers, gives huge optimization in timeframe. This achievement can be used for the bidding sites or related area.

2.2 Feedback control system

Unlikely other (mechanical or aeronautical) disciplines of Engineering, linear control theory [14] is considered in Cloud Computing to automate the dynamic adaptation of resources and take the elasticity benefit of the shared resources. This theory uses input-output relationships of linear systems from the monitoring system: *Stability* (finite inputs produces finite out-puts), *Bias* (how efficiently SLOs are achieved), *Response time* (how quickly the system responds to a change in input) and *Processing time* (how long it takes to reach in steady state) [8].

Control systems are classified as either *Open-Loop* or *Closed-Loop*. An *Open-Loop Control System* is controlled directly by an input signal only, without the benefit of feedback. An example of such system could be an amplifier and a motor. The amplifier receives a low-level input signal and amplifies it to drive the motor to perform the desired job. Open-loop control system is not as accurate as a closed-loop control system [64].

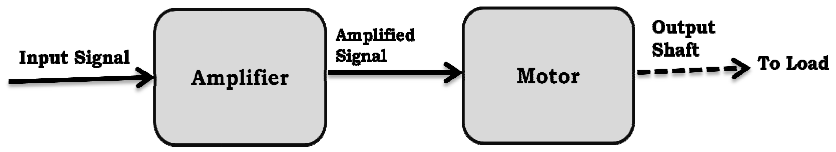


Figure 2.3.1: Basic Block Diagram of Open-Loop control system [64]

As shown in figure 2.3.1 an *Open Loop System* has an input signal that is fed to the amplifier. The output of the amplifier is proportional to the amplitude of the input signal. After amplification, the input signal is fed to the motor, which moves the output shaft (load) in the direction that corresponds with the input signal. The motor will not stop driving the output shaft until the input signal is reduced to zero or removed. This system usually requires an operator who controls speed and direction of movement of the output by varying the input. The operator could be controlling the input by either a mechanical or an electrical linkage.

The concept of *Feedback control* is used to measure the system's outputs such as response time, throughput and utilization to achieve certain goals. As the measured outputs are used to determine the control inputs and the inputs then affect the outputs, the entire architecture is called *Feedback* or *Closed-loop system*. An example of a *Feedback Control System* could be the

Thermostat in a House. A *Thermostat* achieves the desired temperature (output) by adjusting the furnace cycle or fan (input). The desired temperature is maintained even when outside temperature increases or decreases (disturbance).

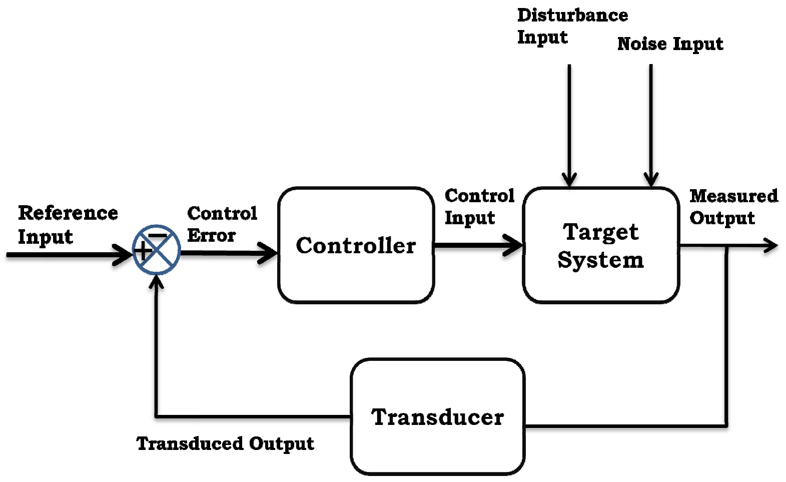


Figure 2.3.2: Block diagram of feedback control system [1]

As seen in figure 2.3.2, the system is a single input and single output (SISO) control system that has single *Control input* (i.e: MaxClients in Apache HTTP server) and single *Measured output* (i.e: CPU Utilization). In this diagram, the *Reference Input* is the desired value of the system's *Measured Output.* The *Controller* adjusts the setting of the Control input to the *Target system* so that *the* Measured Output is equal to the *Reference Input*. The *Target system* is the computing system that needs to be controlled. *Control Error* is the different between Reference Input and Measured Output .*Disturbance Input* is the change that affects the way in which the Control input influences the Measured Output. Noise input is the effect that changes the measured output produced by the target system. The *Transducer* represents the effect such as unit conversion or delay [1].

From the cloud platform aspect, the major challenge is to find out a set of perceptible sensors and actuators to enable control policies to function effectively. Consider the classical controller design methodology that consists of two steps [15]. Design a *transfer function* which relates past and present *Input* values to past and present *Output* values. These transfer functions constitute a model of the system. Then define an *event* that occurs based on properties of the transfer function. Techniques from control theory are used to predict how the system will react when event is generated to it. This control policy can be modified with *Integral control* [13] by using a dynamic target range which decreases as the accumulated actuator values increases. Proportional thresholding can be used for dynamic range while maintaining a certain CPU utilization target [7].

Integral Control can be defined by,

$$U(r+1) = U(r) + K_i * P_{ref} – P_t$$

Where u(r) is the current status, K_i is the integral *Gain parameter* [8], P_{ref} is the current size of the queue, P_t is the threshold. The parameter $K_i > 0$ is called the gain. In this scenario, the increasing size of the request queue will create an event notification to the Worker to process a job as well as maintain higher CPU utilization. To reduce the over provisioning cost of cloud resources, we define P_H and P_L as the high and low of request in the queue, which defines the target range. So, the modified integral control is as follows:

$$U(r+1) = \begin{cases} U(r) + K_i * P_H - P_t, & \text{if } P_H < P_t \\ U(r) + K_i * P_L - P_t, & \text{if } P_L < P_t \\ U(r) & \text{otherwise} \end{cases}$$

This approach will ensure resource provisioning on time by generating specific trigger when the queue is above the high threshold.

3. Policy based scaling

This chapter describes the major aspects of *Policy Based Approach* to provide the elasticity. It will define the basic concept of this scaling, *Event* and several types of *Triggers* that will be generated to provision cloud resources on demand according to SLA. It also illustrates several the *Capacity planning* and the *Optimization algorithms* that were used to ensure optimal usage of cloud resources. At the end of this chapter, resources provisioning technique will be described in details.

3.1 Define Policy Based Scaling

The *Policy based scaling* system enables users to deploy their application into clouds based on SLAs such as deployment deadline period, preferred maximum pricing and response time or the throughput. This scaling method is performed automatically based on the application, platform, and system events. Events can also be generated from the environment in case of deadline period is nearby or price of a resource changes. In this approach, a monitoring system tracks the length of the request queue, response time or throughput of the system and resource provisioning time. Then the optimization process is ensured with the information received from monitoring system using one optimizing algorithm. The Optimization will be performed through management of the workload and dynamic provisioning of cloud resources. Optimization process decides when to send trigger to provision new Cloud resource. It will ensure maximum utilization of Cloud resources by making the appropriate amount of delay if necessary, while provisioning rather than provisioning immediately.

3.1.1 Events

An event is a collection of individual histories which is used to characterize an environmental message transmitted to the system. Events are also called stimuli. [7]. Email confirmation of an airline reservation or a message that reports an RFID sensor reading, are different kinds of events. An event may be time-based, activity-based or derived event which come up after occurrence of another events inside system. In Cloud, a partially ordered set of events either bounded or unbounded where the partial orderings are imposed by the causal, timing and other relationships between the events [8]. In this method, activity such as filling job queue, high CPU utilization, lower storage capacity has been considered as events. Here event will appear in order to a trigger an action.

3.1.2 Trigger Management

A trigger defines an action that should be taken when some events occurs into the system. In database concept, a trigger is a script that executes before or after specific data manipulation language (DML) events occur [75]. Triggers are used to maintain complex integrity constraints and business rules. It is also used to notify system for automatic signaling other programs that action needs to take place when changes occurred. A trigger has two states “Before” or “After”.

This section illustrates several types of triggers that can be sent to create an event in order to provision Cloud resources. A generic concept like following:

<u>Trigger Structure</u>

Create trigger ***triggerName*** on (***ObjectName)***

Condition

trigger_events

Action

IaaS cloud-enabled actions (e.g. deploy new Instance)

3.1.2.1 User Initiated trigger

User Initiated trigger is requested by the user that initiates a request to start or stop an instance on demand at a particular time. This type of trigger is particularly required in any exceptional circumstance or occasion. For example, consider a website that becomes incredibly popular suddenly and receive a large influx with visitors or application that deals with disaster situations (earthquakes, tsunami, etc) could be an example here.

3.1.2.2 Time based Trigger

Time based trigger will be generated to start an instance of an application that needs to run in between a particular timeframe of the day and stop after running certain amount od duration. The Scheduled timeframe have to be agreed by both parties and mentioned in SLA. Provider will issue this trigger as per agreement. For example, a trigger that runs 20 instances for an application between 9.00-17.00 business hours due to large number of user access, but in other time it runs only 2 instances. A trigger that runs an application only 12.00-14.00 hrs which processes restaurant orders during lunch and dinner could be another example of this type of trigger.

3.1.2.3 Pricing Trigger

In the SLA, users have to mention maximum affordable cost to deploy an application into the Cloud. Provider may offer multiple time slots (e.g: Peak and Off-Peak) with variable prices to ensure maximum utilization of resources. Peak and Off-Peak time slot will be defined by the provider as per resource utilization metrics. *Pricing Trigger* will be issued by the provider when the cost of the instance fits to the SLA within deadline. Let's consider a case, where a cloud user can have agreed to pay $50 per month as computing cost for a specific application (e.g:

promotional campaign). Here cloud provider will decide when to start the instance of this application and when to stop cost after analyzing the cost with maximum resource utilization.

3.1.2.4 Infrastructure Trigger

This type of trigger will be generated on the basis of system monitoring information. It's a reactive trigger that will provision Cloud resources on demand. Several system events may occur that will generate this trigger. Events may be occurred based CPU utilization of servers, number of requests per second and the length of the queue or demand for the storage capacity.

- **Increasing Job Queue**

When the request queue is filling up quickly and exceeds the threshold, *Infrastructure Trigger* will be sent automatically to start new instance. Similarly, when the queue is below the tolerance level, it will stop additional instance. For example, when the average number of request to the queue during last 20 minutes is 30% higher than average number of request over the last hour and it already significant by being over 50%, then *Infrastructure Trigger* will be issued to start more instances to handle this excessive workload. Similarly, if average number of request to the queue during last 20 minutes is 50% lower than average number of request over the last hour, stop additional instance.

- **Performance Trigger**

Cloud Provider's Monitoring system will examine CPU utilization metrics or response time to deploy an instance and notify an event when it finds any uneven change in the metrics. This event generates this trigger to start or stop instances to distribute this workload. For example, an event can be set that interrogates average CPU utilization during last 30 minutes is 20% higher than average CPU utilization over the last hour and it already significant by being over 50%, then triggers of scale up initiated automatically. Again, if the average response time during last 30 seconds exceeds 500ms then send this trigger to add additional instance. If the average response time over 30 seconds drops under 100ms then remove instance.

- **Storage Trigger**

Monitoring system will check existing storage usage information. It will create an event to generate trigger whenever additional Storage is required. Provider is responsible to issue this trigger as per your system requirement within user's budget constraints. For example, one user belongs an account to store 1TB data storage into the cloud. However, for a certain application it is necessary to have 1.5TB of storage capacity. In this case, provider sends a trigger to allocate additional capacity for this user. In this case, user will be liable to accept this cost.

3.1 Capacity Planning:

Capacity planning is simpler for an existing system with few adjustments to meet expected changes in capacity. However, planning for a new system requires other prerequisite tasks such as analysis of historical project archives, industry standards and information obtained respective stakeholders [26]. Cloud Infrastructure provider needs to ensure on demand recourse provisioning within budget constraint and maximum resource utilization. To do that capacity planning is necessary on the Cloud provider side. Each instance of Cloud based applications requires resources such as CPU, RAM, Disk (persistent or non-persistent) and Network transfer (in and out). This section illustrates the Capacity planning model which was used in this *Policy Based Scaling*.

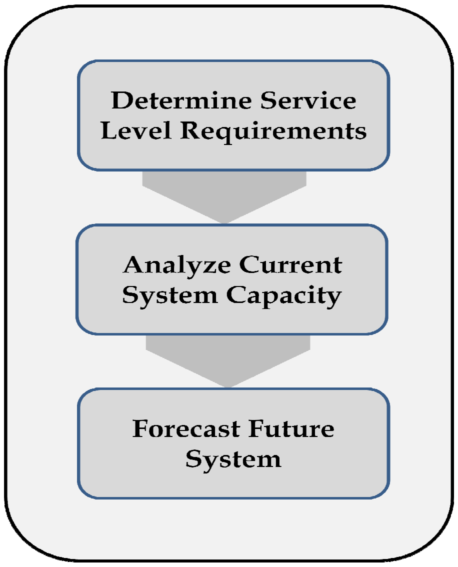

Determine Service Level Requirements
Analyze Current System Capacity
Forecast Future System

Figure 3.1: 3-Steps model of Capacity Planning

3.2.1 Determine Service Level Requirements

The first step is to categorize the type of work done by the system and quantify the user experience as it relates to that work. In this step, providers need to look overall process of establishing service level requirements demands. Before that they need to determine the unit of measuring the incoming work. Finally, establish service level requirements that are supposed to deliver [15].

- **Define Workloads**

In this step of capacity planning, workloads must be defined and a reliable service definition must be created. A *Workload* is a logical classification of work performed on a computing system. Three major issues are need to considered here: “*Who*” is doing the work, ‘*What*’ -type of work is being done and “*How*” the work is being done [15].

- **Determine the Unit of Work**

It is necessary to define an associate *Unit* of work with a workload which is a measurable quantity of the work done against the amount of system resources required to accomplish that work. For example, for an online workload, the unit of work may be a *transaction*, similarly for an interactive or batch workload, the unit of work may be a *process*. [15].

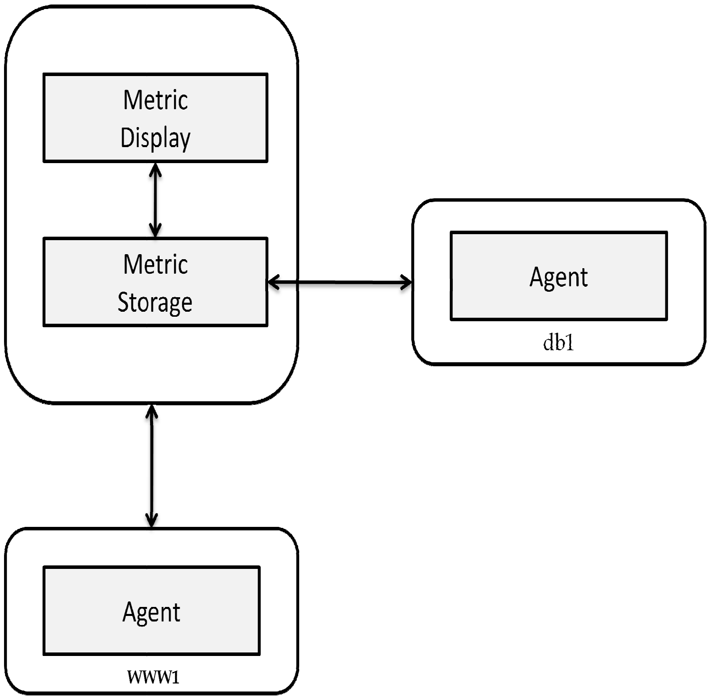


Figure 3.1.2 Fundamental pieces of Metric collection systems [15]

One motivating way is to consider one of the capacity tracing tools, like Metric collection system. As shown in figure 3.1.2, the architecture consists of an “Agent” that runs on each of the physical machine being monitored and a single server aggregates and displays the metrics. When the number of node increases, we need to assign more servers like as data center environment. Agent collects data from the host machine and sends a summary to the metric aggregation server. Then metric aggregation server stores the metrics of individual machine which will be displayed by different manner [26]. Most aggregation servers use database such as *Round-Robin Database (RRD)* [66] in this case.

- **Establish Service Level**

A *Service Level Agreement (SLA)* is an agreement between the service provider and service consumer that delivers acceptable *Services* [25]. This includes items like CPU Utilization by

Workload response time or throughput, processing time for each request and minimum number of requests that can be processed in a given period of time. The workloads help the process of developing SLAs, as it can be used to measure the system according to client requirements. In the case of scheduling application, a *SLA* includes the number of requests that should be processed within deadline. Ideally, *SLA*s are ultimately determined by business requirements based on past experience. To make *SLA*s on present actual service levels, user needs to analyze current capacity before setting service levels. [24].

For example, organizations may want to make reservations to guarantee specific SLA. In that case, SLA needs to be configured on assumption in different tier level [26].

- Tier 1: Allocate 100% of configured memory
- Tier 2: Allocate 50% of configured memory
- Tier 3: No memory reservations set.

The reservation number is defined by the amount of RAM required to run the workload as per SLA. If it is a rule, then need to provision the correct amount of memory for *Instances*. Excessive configuration of RAM on Instances will create an inefficient platform.

3.2.2 Analyze current system capacity

Before planning for the additional capacity, imperative study needs to be performed to evaluate the organizations current capacity. Cloud service providers need to monitor system resource utilization such as CPU utilization, memory, hard drive & network and to find out which workloads are the major users of each resource, where it spends most of the time, allowing one to determine which system resources are responsible for the greatest portion of the response time for each workload. Record and track utilization of system resources to determine where capacity adjustments need to be made to support business processes as defined by the client. In case of having no existing system, information such as historical project archives, industry standards, information obtained from vendors or customer need to consider [23].

- **Measure overall resource usage**

TeamQuest research noticed that CPU utilization is about 64% during time period 7:00 AM - 10:00 AM on a particular day. That means resource seems not to be saturated high memory operations [25]. To determine minimum amount of memory requirement we can consider factor like observing excessive paging in the guest operating system indicates that the *Instance* is not be performing optimally. The exact workload demand is for the *Instance* will ensure adequate reservation [26].

- **Measure resource usage by workload**

CPU Utilization is also a determining factor to measure expected future workload that we chose to treat appointment processes as a workload. Cloud provider needs to set up workloads to correspond to different business activities, thus allow analyzing performance from various requirements from different stakeholders.

- **Identify components of response time**

Components of response time analysis shows the average resource or component usage time for a unit of work. It provides the contribution of each component to the total time required to complete a unit of work. To determine the amount of time which is required to process a unit of work is essential. The resources takes the major response time indicates the concentrate of efforts to optimize performance [26]. *TeamQuest Model* [25] determines the components of response time on a workload by workload basis that can predict which components will be ramp-up in business.

3.1.3 Forecast future system

Cloud provider need to forecast expected workloads for a particular period of time. It is necessary to understand how the changes in workloads affect the business processes and systems for which it was built. Translation of those changes into technical requirements is also necessary to maintain the system at a level that satisfies user demands.

- **Determining of future processing requirements**

In addition to SLA, forecasting of the organization's future is the vital key input into the capacity planning process. Future processing requirements come from multiple sources such as expected growth in the business, requirements for implementing new applications, planned acquisitions, budget limitations and requests for consolidated resources [25].

- **Planning for future system configuration**

To make a suitable plan for future system usage Cloud providers need to create a capacity plan that combines current configuration, required future configuration and the scope to accomplish any necessary system change. They need to monitor and analyze the expected growth rate of the system. The threshold of the queue need to be defined that represents utilization levels and trigger necessary actions to increase capacity. For example, when running instances CPU utilization reaches 90% of its capacity, need to start additional instances. Incident plan take an actions in response to identified capacity triggers to meet the capacity.

Capacity planning often requires tradeoffs to accommodate capacity limitations, quality issues, budget concerns, etc. However, these tradeoffs have to be mentioned in SLA and agreed upon by the both parties. A provider has the scope to choose one of the three strategies while doing capacity planning [25].

- Lead strategy: adds capacity in anticipation of an increase in demand.
- Lag strategy: adds capacity after demand has increased beyond existing capacity.
- Match strategy: adds capacity incrementally in response to changes in demand

3.3 Optimization & Algorithm

Event based workload management system offers user to deploy their application into clouds based on SLAs such as deployment deadline period, preferred maximum pricing, optimized power consumption etc. Optimization will be performed through the management of the workload and providing the dynamic resource provisioning into cloud. This section depicts the optimization process and algorithms that were used to provision the optimal number of cloud resources.

3.3.1 Static Vs Dynamic Scaling

Static Scaling determines the minimum number of instances required to achieve maximum availability at the peak workload and then runs all of those for the desired duration. It provides the best case result for a static allocation that achieves near to 100% availability since there is prior knowledge of the number of instances required. However it is quite difficult to determine the exact number of instances that is needed. In this scenario, users need to maintain multiple times the instances than required to ensure that they will be able to serve their customers even when they have heavy traffic. Though static provisioning ensures maximum availability, it takes higher cost to ensure that. *Static Scaling* is considered as *Vertical Scaling [2]* which provides the facility to change assigned resource into a running instance such as adding more physical CPU for the running instance. It scales the resource through resizing and replacement of the instances. However, the most operating systems do not support changes without rebooting on the available CPU or memory to support this *Vertical Scaling*.

Dynamic Scaling of Cloud resource is the most significant feature in Cloud Computing which is also defined as *Horizontal Scaling* [2]. *Horizontal Scaling* allows adding new server replicas and loading balancers to distribute the workload among all available replicas. As shown in figure 3.3.1, it provides two scheme *Virtual Machine (VM) replication* and *Network Scalability*. *VM replication* will be delivered by using the *Load Balancer Algorithm* along with *Load Balancer Scalability*.

On the other hand, *Network Scalability* is the scalability of Cloud network that provides on demand creation of *Virtual Network* [48] by instantiating bandwidth provisioned network resource. It also provides *Network Slicing* that keeps the application flow separate by adapting on demand network utilization of each application and dynamically allocates network bandwidth [2]. In this policy based approach, scaling will be ensured through the workload *Optimization Algorithm* which will decide when and how much Cloud resource need to provision. After analyzing the monitoring information (e.g : Queue Length, Response time) and budget constraints , provision trigger will be sent.

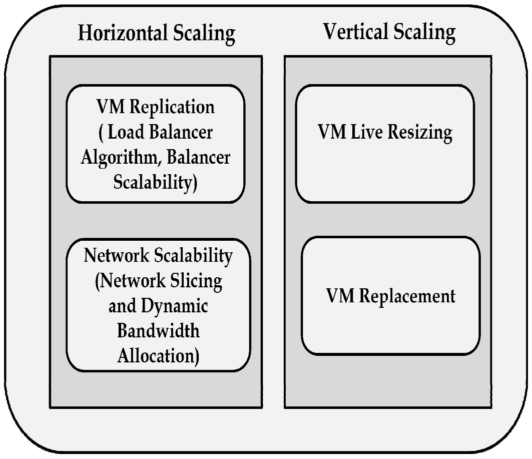

Horizontal Scaling
Vertical Scaling
VM Replication (Load Balancer Algorithm, Balancer Scalability)
Network Scalability (Network Slicing and Dynamic Bandwidth Allocation)
VM Live Resizing
VM Replacement

Figure 3.3.1 Horizontal and Vertical Scaling

3.2.2 Energy Optimization

Determining *Virtual Machine (instance)* power consumption and its operating costs is one of the key factors in the *Policy based scaling Approach*. Removing unutilized resources with this auto scaling method can reduce power consumption and resulting *CO2 Emissions* significantly. On demand Cloud resource provision and customized configuration settings from each user make this challenging to achieve a greener computing environment. A cloud application request instances with a configuration setting that contains type of the Processor, Operating System and installed *Middleware solution* [71]. All of these resources consume different amount of power. For example, the average power consumption of Amazon EC2 cloud infrastructure with five different types of processors, six different memory configuration and nine different OS types is 150-610 Walts per hour [62]. This section describes the challenges of capturing VM configuration and using this information how to optimize the power consumption.

Selecting the appropriate type of instance configuration to run is one of the major challenges for energy optimization. When an application requests an instance which is available in the provider resource pool, the request can be served almost instantaneously. However, if it is not in the pool, more efficient way would be to modify the configuration of a running instance rather than provisioning and booting an instance from the scratch to reduce the *Energy* consumption. Another challenge is to determine the number of available resources in the pool to minimize the *Energy* consumption and their maintenance cost. In addition, each individual configuration in the pool varies in *Energy* consumption and cost. So it is quite challenging to navigate tradeoffs between *Energy* consumption and the response time of the dynamic scaling of different sets of instances configuration.

In the *Policy based scaling* considered that Cloud user provides a configuration *of* the *Demand Model* that describes the setting for each type of instance that the application needs during its execution lifecycle. Every new request for an instance must contain a *Demand Model*. It is a text based domain specific language that describes each configuration requested in the configuration model. Cloud Service Provider belongs the configuration *of Adaptation & Energy Model* that specifies the time required to add or remove an instance as well as the power consumption required to run an instance as per *Demand Model.* The cost of the power consumption will be calculated in the provider side. Then the combination of *Demand Model*, *Adaptation & Energy Model* and the *Workload Estimation Model* are used to derive the optimal scaling setup.

The objective function of the power consumption reduction and maintenance cost has derived from the *Response Time* and the instance *Demand Model* configuration. The *Response Time* R_t, can be defined from the configuration *Demand Model* as:

$$R_t = \min (T_0, T_1, T_2, \ldots T_n, (STRUP (R_q))$$

Where, R_t is the expected response time , n is the total number of feature in a Demand Model, T_i is the Expected request completion time , STRUP (R_q) is the time to start a new instance to fulfill the request R_q.

The expected response time, R_t would be the fastest time available to complete the request. The time to complete the request (T_i) will be zero if the configuration already exists in the resource pool. In other case, the time to complete the request is equal to the time needed to modify the configuration as per request.

Now for each instance configuration K_i, energy consumption can be defined as:

$$Energy\ (K_i) = \sum_{j=0}^{n} qij\ Ej$$

Where Q is a set of features that describes the selection state of each instance configuration and E is the energy consumption cost model resulting from the feature in a running instance configuration.

Each set of variables, $K_i \in K$, describes the selection state of the features for an instance in the queue. For each variable, $q_{ij} \in K_i$; if $q_{ij} = 1$ in a derived configuration, it indicates that the j_{th} feature has selected by the i_{th} instance configuration.

So the overall *Energy* consumption minimization objective function is defined as

$$\varepsilon = \mathit{Energy}(K_0) + \mathit{Energy}(K_1) + \mathit{Energy}(K_2) + \ldots\ldots + \mathit{Energy}(K_N)$$

By using *Virtualization [69]* and Consolidation, the energy consumption is further reduced by switching-off unutilized servers. This scaling approach mostly focused on the inclusion of overall carbon efficiency of all the Cloud providers in scheduling and resource provisioning decisions. If Cloud users have flexible deadline and don't have urgent scheduling of the applications, then datacenters can be run at higher energy efficiency and carbon gain can be achieved. Moreover, this optimized power consumption policy also gives better return to the Cloud Service provider.

3.2.3 Fitness function

The fitness function needs to ensure the desired optimization goal by maximizing the application throughput while minimizing the cost. When Cloud providers allocate required resource to the user application, they need to consider the system throughput. This section illustrates the fitness function of the *Policy based scaling* method.

Assume that resource is allocated the from provider resource pool (R) to the worker (W). The role of a worker can be defined with the function *role(w)* for $w \in W$.

$$\text{role}(w) = \textstyle\sum_i w_{ti}.\text{task}_i, \text{, where } w_{ti} \text{ is the weight of the task i.}$$

Each resource has an estimated performance for different roles which is calculated by weighted sum of its performance on each of the task. The estimated performance of resource *r* for the role *role(w)* can be denoted as capacity(role(w), r).

Each resource has a fixed cost independent of the role. The challenge is to assign the best fits worker to a resource that maximizes the performance of the application and minimizes the cost. The throughput of a worker is the minimum capacity of the resource that is assigned for particular role.

$$\text{throughput}(w) = {}_{w \in W}\text{Min} \left\{ \begin{array}{l} \text{capacity}(\text{role}(w)) \\ \sum w_t.\text{task} \end{array} \right.$$

So the throughput of the system will be as:

$$\text{throughput}_{system} = \text{throughput}(w) \mid \text{role}(w)$$

The cost of the application is the sum of assigned resource cost.

$$\text{Cost}_{system} = {}_{w \in W} \sum \text{cost}(\text{role}(w))$$

Therefore the fitness function of the system can be defined as

$$\text{Fitness}_{system} = \text{throughput}_{system} \; / \; \text{Cost}_{system}$$

3.2.4 Policy based scaling Algorithm

In this algorithm, Feedback control model (explained in section 2.3) was used to maximize the cloud resources utilization while satisfying the *Deployment Time Constraint*, *Resource Budget*, *Response Time* and *Power Consumption*. Here Virtual Cloud Resources (Virtual Machine (instance), Memory) are dynamically provisioned according to the environment conditions & parameters. The major challenge is to finish all the submitted requests before user specified deadline in SLAs within Budget Constraints.

In this control model we have several components such as *Maximum Resources Cost* (C_{Max}) that is affordable by the Client, Application Deployment *Deadline* (T_d) which mutually agreed in SLA between the both Cloud User and Provider, *Average instance provision time* is T_{RP} , *Average Start Up delay* d_{RP} and the successful deployment after considering all factors denoted as $MIF_{success.}$.

If we define the total *Workload* W_N as Vector, then

$$W_N = (J_i , n_i)$$

Where each Job group (J_i) has n_i number of submitted jobs.

The *Computing Power* (P) of an instance I_j can be represented as a vector. *Computing Power* indicates that how many jobs from a job group can be successfully completed before the deadline arrives. Assume that all jobs in the queue will be finished with current instance. So, the *Deadline* and *Individual Job Completion Time* ratio will be as following:

$$P_i = (J_i , \frac{T_d * ni}{\sum T_{RP} * ni})$$

Where n_i is the number of submitted jobs in the queue and T_{RP} is the Average Processing time.

So the computing power of the *Pending instance* can be calculated as,

$$P_i = \left(J_i , \frac{T_d - (d_{RP} - fi) * n}{\sum T_{RP} * n} \right)$$

Where d_{RP} is the average start up delay and fi is the spent time after the instance started.

So the total computing power at ith time stamp $\sum P_i$.

If $\sum Pi < W_N$, then more instances need to provision to distribute additional workload. At this time, optimization process needs to find one of the best fits instance with Computing Power

$$P_i^1 = W_N - \sum P_i$$

and the cost would be $Min(\sum Cinew)$, but with maximum computing power, Max ($\sum P_i^1$) to optimize the resource budget.

So the cost of the new instances would be within budget constraint

$$\sum Cinew + \sum C_i <= C_{Max}$$

Where C_{Max} is the maximum Budget for a resource and C_i is the cost of an instance .

Since VM Instances are charged to User based on operating hours even though full hour may not be used, additional instances need to shut down before full hour completion. So, the computing power without that instance needs to calculate and compare with current workload.

$$\sum P_i - P^1_i > W_N$$

Considering the Energy consumption factor described in section 3.2.3, total energy must be

$Min(\sum \varepsilon_{I\ type})$, where I_{type} is the instance type.

So, the success of the Management Influencing Factor (MIF) can be defined as

$$MIF_{success} = \sum C_i + \sum Ci_{New} <= C_{Max} , \sum T_{RP} + \sum d_{RP} <= Td , Min (\sum \varepsilon_{Itype})$$

Algorithm 3.2.3 Cost Function for an Instance Vector considering the price of minimal instances

```
Procedure Cost ( instances[ ],minInstanse[ ], C [ ])
    $C_{inew} \leftarrow 0$;
    For i ← 0; i< instances[ ] do
        If  instance [i ] ≥ minInstanse [i] then
            $C_{inew} \leftarrow C_{inew}$ + instance [i ]  · C [i];
        Else
            $C_{inew} \leftarrow C_{inew}$ + minInstanse[ i]  · C [i];
        End if
    End for
    Return Price
End Procedure
```

Algorithm 3.2.4 Optimization of Deadline, Budget & Energy Constraint

```
Procedure  Optimization ($Ins_{current}$ , C [ ], $Ins_{Type}$,  $T_{RP}$ ,  $d_{RP}$ ,  $n_i$ , Energy )
    instances [ ];
    int Threshold;
    while ($n_i$ >= Threshold) do
        $C_{temp}$ = Cost  ($Ins_{current}$, minInstance[ ], C [ ]
        Energy = $Ins_{type}$ . $Ins_{current}$;
        If ($C_{temp} < C_{Max}$  && $T_{RP} + d^{RP} < T_{Deadline}$ && Energy <=Min(Energy)  )
            Send Provisioning Trigger;
            $Ins_{current}$ ++;
        End if
        Return  int $Ins_{Req}$, int $Ins_{current}$;
    End  While
End Procedure
```

3.4 Resource Provisioning

According to the *Policy based scaling* approach, the Cloud service provider is responsible to provision resources. Client can only request for additional resource, but provider will decide when to provision. Provider is liable to provision resources as per SLAs. The decision of resource provisioning will be controlled by changing the configuration file in provider side.

The process *Monitor* keeps track of historical job processing time, request arrival pattern and startup delay of the instance. By maintaining this historical information, *Optimizer* can decide to send best fit provisioning trigger to prepare for possible workload surges early. Optimizer has its own algorithm that analyzes environment conditions while sending a trigger. *Provisioning manager (PvM)* works as the adapter between *Optimizer* and cloud Provider. *PvM* allocated resources as per the decision from *Optimizer*. For example, Amazon EC2 instances are grouped into three types: Standard, high-CPU and High-memory. Standard instances are suitable for all types of applications where as High-CPU instances mostly suited for computing intensive application such as image processing and High-memory instances are dedicated to I/O intensive applications [21]. Similarly, based on the type of the application, *PvM* starts a new instance to the user.

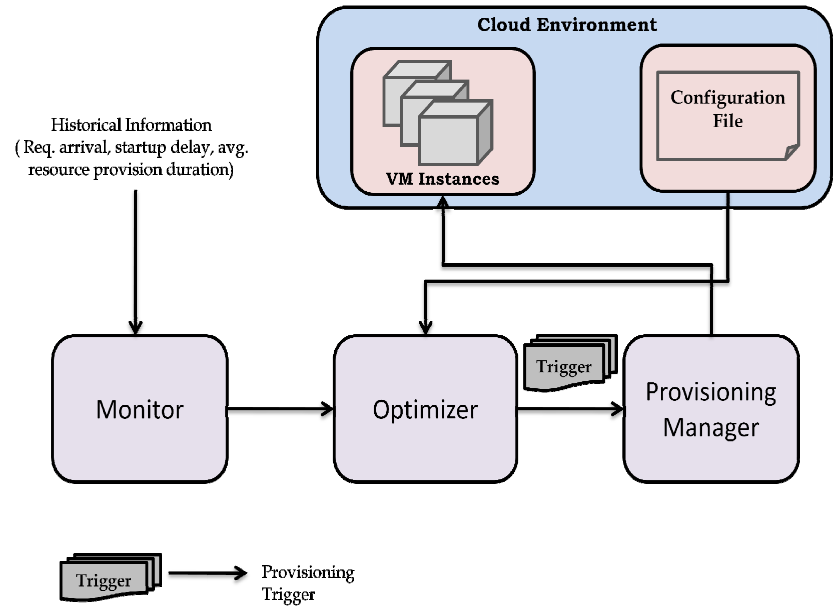


Figure 3.8: Cloud Resource provisioning

As seen in figure 3.8, Optimizer is the core of this scaling mechanism. Depending on real-time workload and historical job information from performance monitor and configuration parameters set in configuration file, it sends a best fit trigger to the provisioning Manager (PvM) to start an instance. After getting notification from *Optimizer*, *Provisioning manager* determine the instance startup plan. In contrast, releasing instance actions are initialed it decides which instance is approaching full hour operation and could be the potential shut-down targets. In this case, *PvM* notifies *Optimizer* to check whether remaining computing power is large enough to handle the workload.

3.6.1 Deadline Accomplishment

Satisfying deadline is one the major MIF factors in this Policy based approach. When workload have Changed but deadline is fixed, PvM follows the *Earlier Deadline First Schedule (EDFS)* [67] method that allows finishing jobs those are approaching to deadline. If necessary, it provision new instance to handle this excessive workload. When the job has completed, it shut down the additional instance. On the other hand, when workload is fixed, but user initiated trigger has sent (request on demand), PvM first reset the job provisioning priority according to deadline. Then it schedules the higher priority job at first and then others.

3.6.2 Peak and Off-Peak Slot

Whenever in SLA cloud user put lengthy timeframe as deadline, provider has the scope to move the application deployment into cloud in one of the suitable time slots. For example, large volume data processing applications which need data computation and analysis can be performed during day time, whereas data backups and movements can be shifted durings nights and holidays according to SLA conditions.

Cloud user will define the types of application that shall be deployed into the Cloud. This scaling mechanism will schedule computing intensive applications such image processing and scientific computation during Peak hours (8.00-20.00 hrs) using High-CPU instance within budget constraints since most of the scientific tasks are made this period. Similarly, I/O intensive or memory oriented applications like backup or storage services can be served in off-peak hour using high-memory instances. Off-peak hours will be duration between 21.00 – 8.00 hrs. The peak and off-peak slot is dependent of data-center location such as US & EMEA has different time zone. These time slot will be defined by cloud provider and every provider have their own peak and off-peak slots as per their request traffic.

3 Implementation

This chapter describes the implementation technique of the Policy based scaling approach. It provides the system architecture as well as the scenario of the test environment. The prototype of this Policy based scaling system was implemented on Amazon Elastic Cloud platform which also includes few other Amazon web services platform like as Simple Queue Services (SQS) and Auto scaling. The test bed was developed on Java eclipse environment.

4.1 Client-Worker Paradigm

The traditional *Client-Worker paradigm* [68] was considered to implement the thesis concept as seen in figure 4.1.1. In this implementation Client was renamed as “Requester” and worker as “Worker”. Requester sends message to the Amazon SQS Queue (described in next section) which can be granted as client request for cloud infrastructures services. The Worker will read the message from the queue. The worker is configured with the same queue as the requester uses for sending.

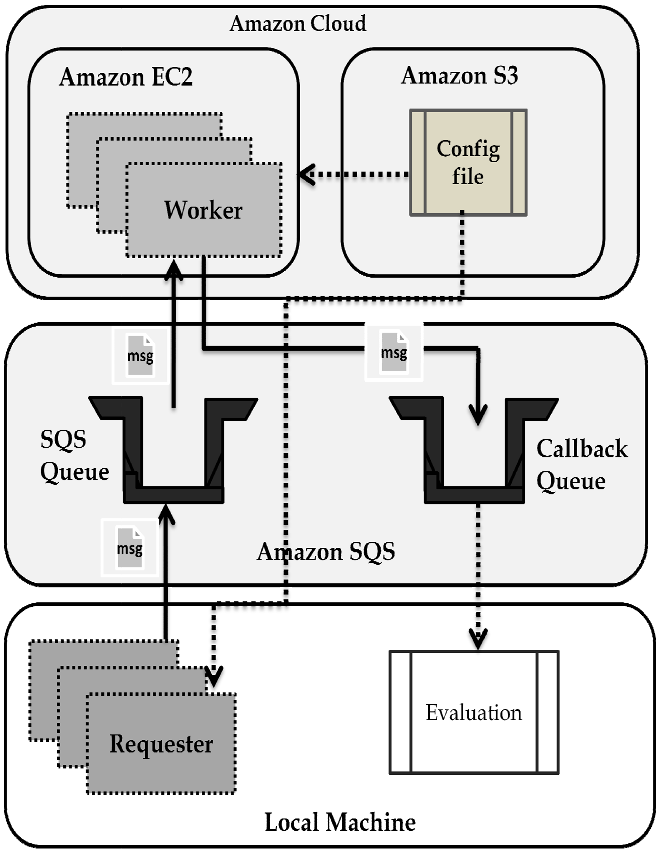


Figure 4.1.1: Implementation Scenario

In the implementation scenario, both Requester and Worker have *Configuration File* shown in figure 4.1.2 which is stored in Amazon S3 storage. The *Configuration File* contains one parameter: no of message per minute. Upon initialization, Requester creates its own, random Amazon SQS queue which receives reply message from Worker. When the requester sends a message, it includes the response queue as well as a unique message id. After a while, the response queue receives a processed message with the same id and writes the message processing time along with ID to a log file.

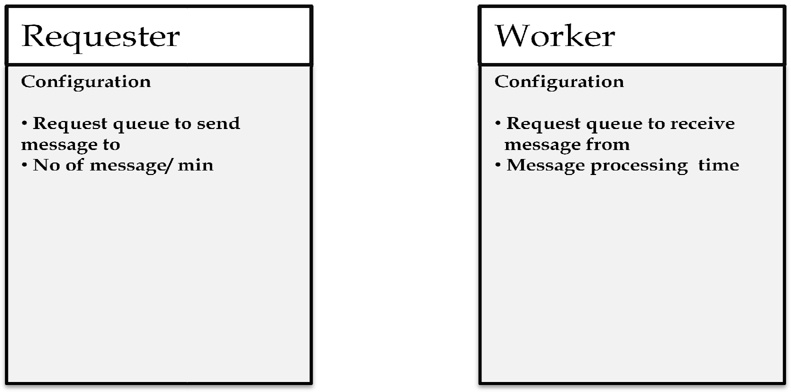

Requester
Configuration
• Request queue to send message to
• No of message/ min
Worker
Configuration
• Request queue to receive message from
• Message processing time

Figure 4.1.2: Requester-Worker Configuration File

The Worker considered SQS as input queue. It has also one configuration parameter: Processing time (the amount of time it takes to process a message. When it receives a message from the requester, the worker processes the message (waits amount of time mentioned into the configuration file) and sends back a confirmation message to the respective "Callback Queue" of the requester. The test result (message processing time and Message Id) is stored in a log file through the Callback Queue.

When a client invokes a two-way asynchronous operation on an object, it passes an object reference for a reply handler as a parameter. The reply handler object reference is not passed to the worker, but instead is stored locally by the client. When the Worker replies, the client receives the response and dispatches it to the appropriate callback operation on the reply handler provided by the client application. In this implementation seen in figure 4.1.1, Callback queue is also a SQS Queue which sends the confirmation message to the requester and writes ID and respective processing time in a log file.

The Worker is one of the AMIs (Amazon machine image) inside EC2. Whenever the number of the incoming requests to the queue is very high, it creates overload to the existing workers, an event occurs as per monitoring information to send a provisioning trigger to the provisioning manager to start additional worker. Similarly, as per monitoring information provisioning manager stops additional worker to reduce infrastructure cost, power consumption and optimize maximum usage of cloud resources.

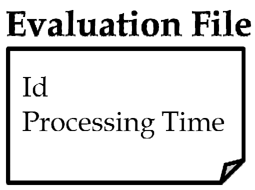


Figure 4.1.3 Evaluation file

As per configuration constraints, AMI Worker takes certain amount of time to process a message. During this period, it reads a message from the queue, delete this message from the Queue and send back the confirmation message to the respective "Callback Queue" of the requester. The callback Queue writes the result of the execution into a text file which called "Evaluation file" as shown in figure 4.1.2. It contains Message ID and relevant message processing time.

4.2 System Architecture

4.2.1 Monitoring

In this approach, monitoring plays one of the vital roles behind cloud resource provisioning. We will monitor incoming requests in the queue from Cloud users. Monitoring is ensured through following monitors:

Request Monitor: The Request Monitor mechanism keeps track of the execution status of client requests, in other way it informs about size of the queue. Messages which are visible into the Queue have different levels. As seen in figure 4.2.1, "Tolerance level" is the first level that can be handled by existing Workers. When the queue size exceeds threshold, an event occurs to alarm the Trigger management. As it reaches the firing point, immediately sends a signal to the Trigger management module to start more workers.

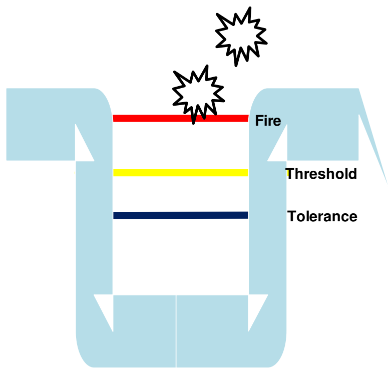


Figure 4.2.1: Request Queue status

VM Monitor system monitors the availability of VMs and their resource entitlements. It keeps the history information of resource provisioning duration as well as stopping additional Resources.

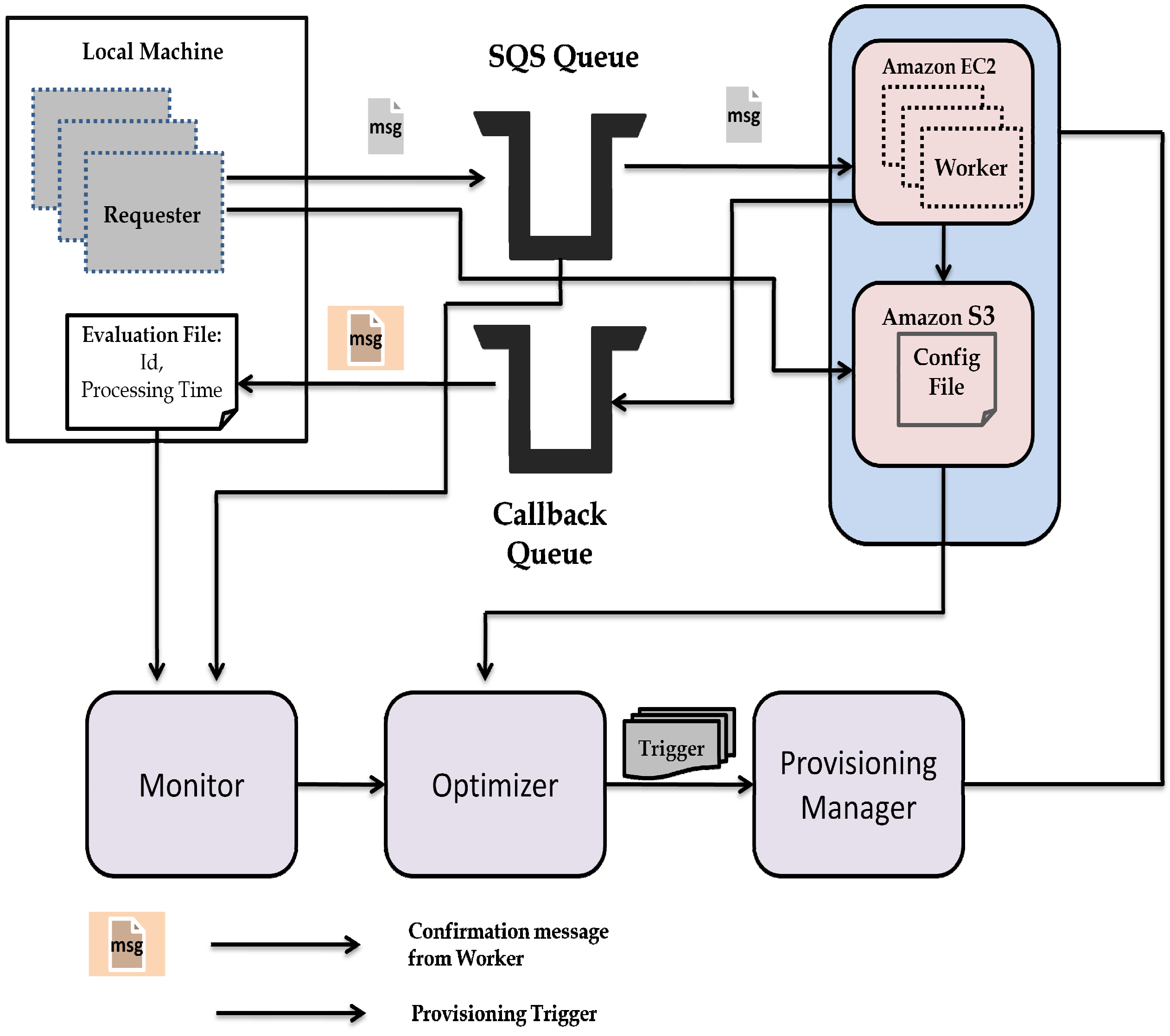


Figure 4.2.2: System Architecture

4.2.2 Optimizer and Trigger Management

Optimizer analyzes the Management Influencing Factors (MIFs) such as provisioning or request handling time behind each request. It also considers price, deadline, power consumption, network usage and similar cloud infrastructure resources. If it fits with the surrounding conditions, triggers will be sent to provision additional cloud resources.

Cloud providers allow user to request more resources. Unlimited resources scale applications to extremely large size, but need to pay for every cycle used and byte transferred to cloud. In this scenario, *Optimizer* selects the payment method (described in sec 3.3) to provision new instance.

Cloud instances acquisition requires certain amount of time since acquisition request come to make the instance available. *Optimizer* process collects previous start up information from monitoring and predicted average start up time to prepare early for workload surges. Considering all the monitoring & environment constraints, *Optimizer* sends one of best fit provisioning triggers (seen in sec. 3.4) to request a resource. In this implementation seen in figure 4.2.2, the trigger request for an EC2 AMI as worker to distribute extended load among multiple workers. *Optimizer* also provides auto scaling mechanism to avoid the fractional billed amount of VM instances as EC2 consider 10 minutes and 60 minutes as equal time. It select the best suitable EC2 instance among all three (described in section 4.2.3) to deliver the most cost- and energy effective service.

4.2.3 Provisioning Manager (PvM)

After receiving provisioning trigger from Trigger Management module, *Provisioning Manager* (PvM) execute VM (Virtual machine) start up plan. In this implementation, PvM starts an Amazon AMI instance (Worker) if necessary and shut down to optimize budget constraints.

Amazon EC2 provides two types of booting up of an instance. Boot from an S3-backed AMI known as ***Instance store AMI*** and other one is ***EBS-backed AMI***. Each boot up delivers different life cycle models. Average instance startup time in Amazon is 10 minutes, however also changes time to time. However, VM shut down time is more likely stable, usually 2-3 minutes [21].

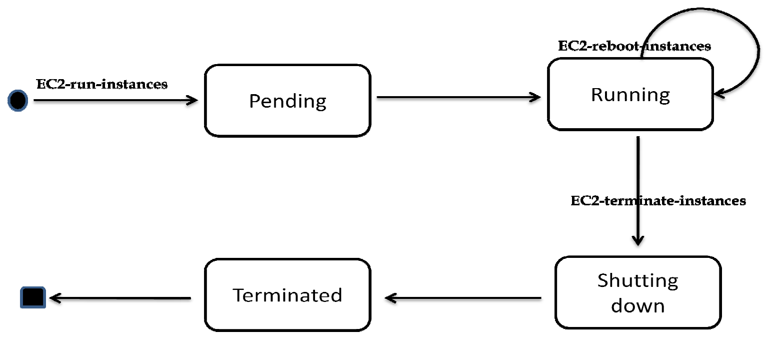


Figure 4.2.4: S3-Backed AMI Instance [19]

As shown in figure 4.2.4, life cycle of the *S3-backed AMI instance* starts as we lunch it. It stays little time in “pending” state and waits for the reservation. As soon as it begun to boot, “running” state starts that continue until giving terminating command. Before terminating, instances wait little amount in “shutting down” state.

In contrast, *EBS-backed AMI instance* shown in figure 4.2.5 also begins its' life when launched, spends little time in the "pending" state until moves to "running" state. From there it can be rebooted or shut down, like S3-backed AMI instances. In addition, it can also "stop" and "start" again or terminated. EBS-backed AMI instance is more expensive than S3-backed. Usually, EC2 instances are not charged when it is in "stopped" state, but the attached EBS volumes cost money. EBS-backed AMI instance deletes the associated EBS boot volume when it is terminated only, so the cost of the EBS boot volume is there [20]. *Provisioning Manager* considers the pricing issue of *EBS-backed AMI instance* when launching an instance. Application which has higher budget and need to reboot within short duration *EBS-backed AMI instance* can will be more suitable. In other case, *S3-backed AMI instance* will be selected by *Provisioning Manager.*

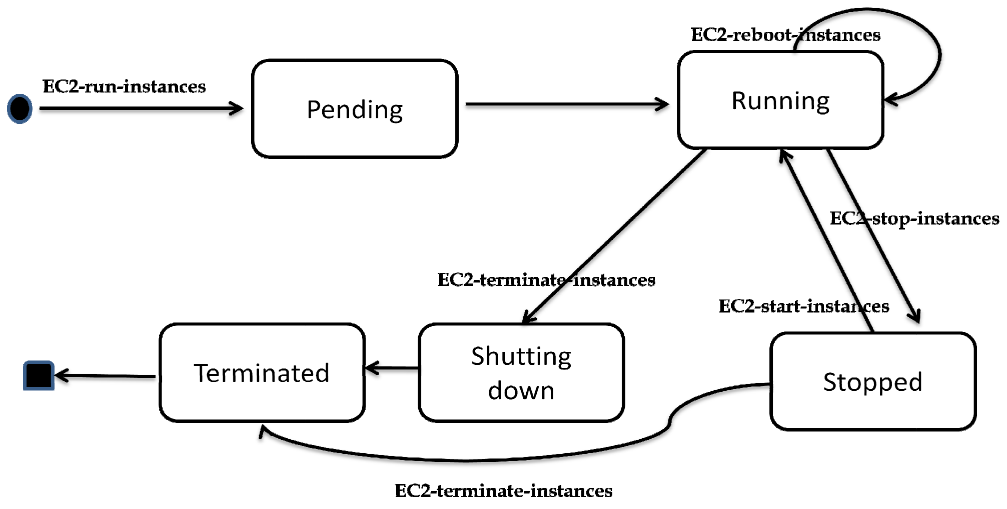


Figure 4.2.5: EBS-Backed AMI Instance [19]

4.3 Asynchronous Messaging with Amazon SQS

Unlikely other asynchronous messaging system, Amazon Simple Queue Service Queue follows Fire-and-Forget information exchange process. Sender of an asynchronous messaging system does not need to wait for a response from the recipient since messaging infrastructure ensures the delivery of the message. It is also one of the vital components in *Loosely Coupled Systems* where the components in a system can work together without being dependent to each other, particularly in web services. SQS also allows participants to communicate reliably even if one of the parties is temporarily offline, busy or unavailable.

4.3.1 SQS Message Lifecycle

A single message size in SQS is limited to **8K**. SQS ensures at least one delivery of message, multiple is also possible. User can create unlimited number of queues with one account.

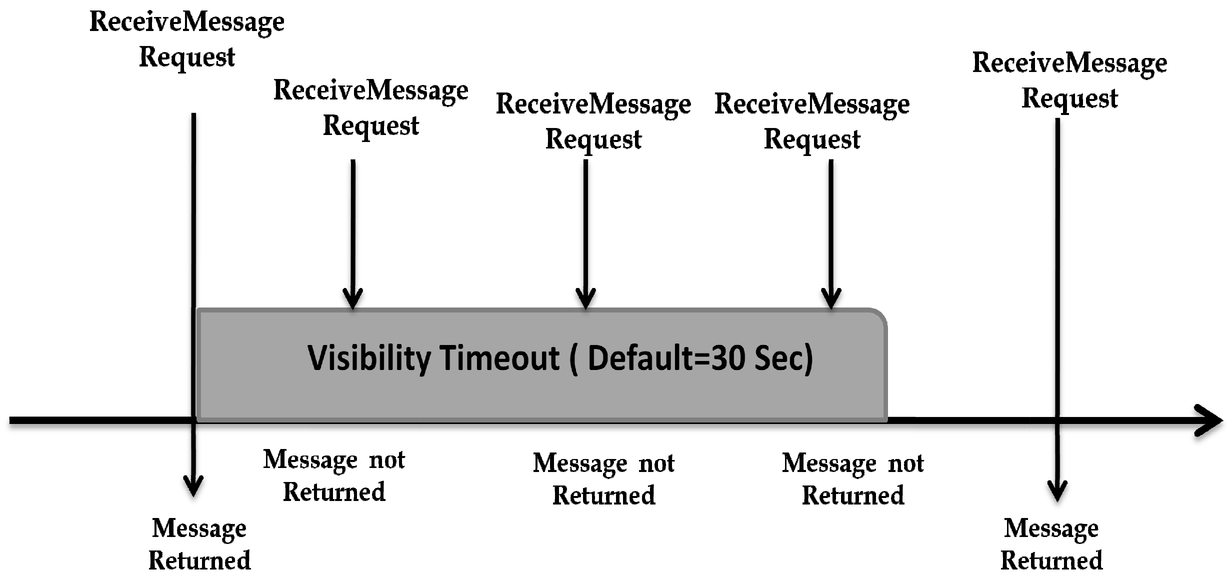
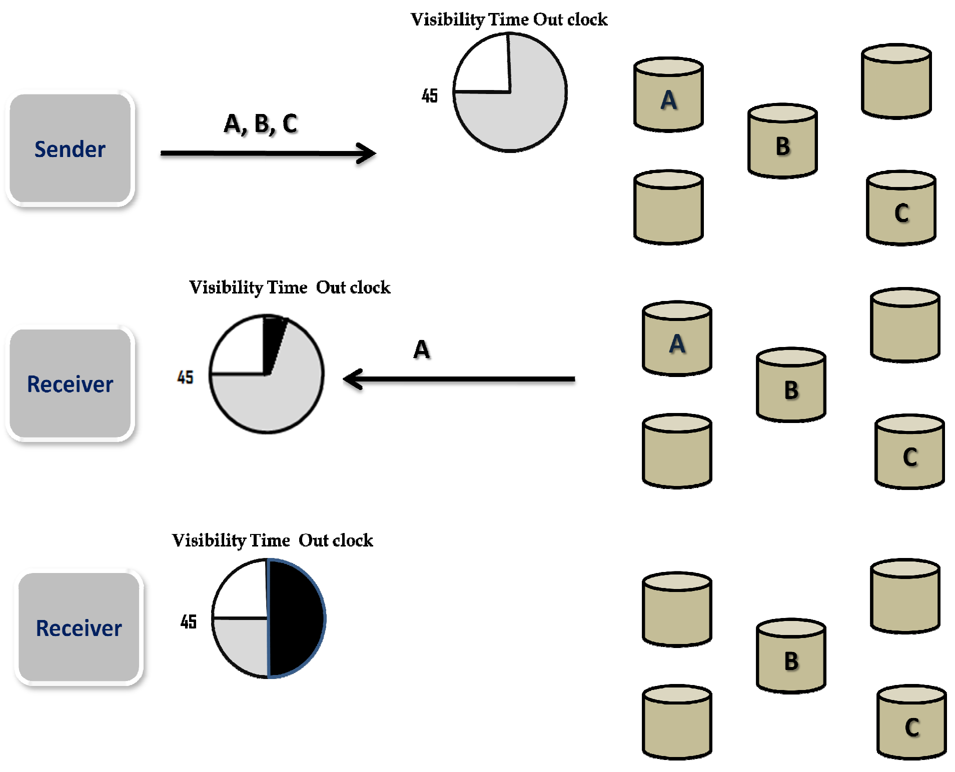


Figure 4.3.1: Message Visibility Time out [18]

A *Received Message Request* is the request from the receiver to receive the message that will not be returned until the *Visibility Timeout* has passed. *Visibility Timeout* is the duration that a received message from a queue will be invisible to other receiving components when they try to receive messages. Messages those successfully finished reading need be deleted. Otherwise, Amazon SQS blocks with a visibility time out, period of time during that Amazon block message to avoid further reading and processing by another process. Default visibility time out for a message in SQS Queue is 30. User has the opportunity to change Visibility time out for the entire queue. Typically, the visibility timeout is the average time it takes to process and delete a message from the queue [18].

4.3.2 SQS Functionality

SQS message send-receive monitoring process and Load balancing algorithm indicate that, SQS Queues are not FIFO (First-In-First-Out), rather it shows the randomization of message processing [17]. AWS scalability mechanism redirects messages to different servers. Two messages from same application can be stored in two different servers. Amazon load balancing and scalability mechanism may redirect the receiver to receive messages first from Server 2 and then from Server 1 which results delivery of the messages in random order.



Visibility Time Out clock
45
Sender
A, B, C
A
B
C
Visibility Time Out clock
45
Receiver
A
A
B
C
Visibility Time Out clock
45
Receiver
B
C

Figure 4.3.2: SQS arbitrary Message processing and Visibility time out [17]

As seen figure 4.3.2, sender sends message A, B, C to a queue and the message is randomly distributed any of the across the SQS servers. Receiver retrieves messages from the queue, and message A is returned. While message A is being processed, it remains in the queue and doesn't send subsequent receive requests during visibility timeout duration. Receiver deletes Message A from the queue to avoid the multiple received and processed of this message after visibility timeout expires.

4.3.3 SQS Queue Implementation

To use Amazon Web Service properties, every user need to have an authorized AWS account. An AWS account consists of secured formation such as User name, secured password, Access Id key & Secret Access key**.** Among all features, followings were used to implement Requester-Worker message sending scheme [17].

CreateQueue

It Creates queue using authorized AWS account in any of the Amazon distributed server. Queue name is defined by the unique URL. In this implementation, it was used to create the request queue "SQS Queue" and the response queue "Callback Queue".

List 4.3.3 : SQS Create Queue Implementation

```
Public static String createQueue( String queueName) {
        sqsService =new AmazonSQSClient(new
        PropertiesCredentials(MainRequester.class.getResourceAsStream("AwsCredentials.pr
        operties")));
        msgQueue= new CreateQueueRequest(queueName); // request to create the queue
        url= sqsService.createQueue(msgQueue).getQueueUrl(); // URL is the unique identifier
                      of the queue which is used to send and receive message from the queue.
        return url;
        }
```

SendMessage:

In this implementation, it was used to send a message request to a specified queue URL to add a message and stored it one of the distributed Amazon Server.

List 4.3.4 : SQS Message Sending

```
Public synchronized void sendMessage(SQS_Message sqsMessage, String
myQueueUrl ) {
          String msg=createMessage(sqsMessage);
          System.out.println(this.getClass().getName()+ "Message sent to:"+
          myQueueUrl);
          System.out.println("Message:"+ msg);
          SendMessageRequest smr=new SendMessageRequest(myQueueUrl,msg);
  sqsService.sendMessage(smr);
  }
```

ReceiveMessage

ReceiveMessage function was used here to return one or more messages from a specified queueUrl.

List 4.3.5 : SQS Receive Message

```
public  SQS_Message recieveMessage(String myQueueUrl) {
            SQS_Message recievedMsg= new SQS_Message();

            Message message=null;
            message = sqsService.receiveMessage
            (receiveMessageRequest).getMessages().get(0);

            System.out.println(message.getAttributes().size()); //  this line shows the
                            number of attributes that included with the message
            return recievedMsg;
```

DeleteMessage

In this implementation, it was used to remove a message from a particular queueUrl after reading. ReceiptHandle The receipt handle associated with the message to delete it from the Queue.

List 4.3.6 : SQS Delete Message

```
public DeleteMessageRequest(String queueUrl, String receiptHandle) {
                sqsService.deleteMessage(new DeleteMessageRequest().
withQueueUrl(myQueueUrl).withReceiptHandle(amazonMessage.getReceiptHandl
e()));

// Receive Handler the unique acknowledgement received from the receiver and
used to delete messages from the queue

}
```

4.3.3.1 PrintQueueList

To Prints the list of QueueUrl of one or more Queues this method is used. In this implementation, it prints the list of the queue which has already created.

List 4.3.6 : SQS PrintQueue List

```
Public static void printQueueList(AmazonSQS sqs) {
for (String queueUrl : sqs.listQueues().getQueueUrls()) {
        System.out.println(" QueueUrl: " + queueUrl);
        }
}
```

4.3.3.2 GetQueueAttributes

GetQueueAttributes were used to show the queue attributes such as ApproximateNumberOfMessages and CreatedTimestamp. *ApproximateNumberOfMessages* returns the approximate number of visible messages in a queue and *CreatedTimestamp* returns the time when the queue was created. In this implementation, Worker put a timestamp when it read the message from the request queue and send the confirmation to the response queue.

5. Evaluation

5.1 Workload Scenarios

This section demonstrates different workload scenarios on test case environment in local machine as well as in Amazon EC2 Cloud. It will also analyze test case results with a variety of optimization algorithms.

5.1.1 Test case 1 : Static Load

In the figure 5.1.1 One Requester and one Worker were considered during the entire execution period. Requester send message to the Amazon SQS queue in a static manner (one message in every 10sec). Since there is no change to the worker, the worker takes nearly same processing time to finish one request.

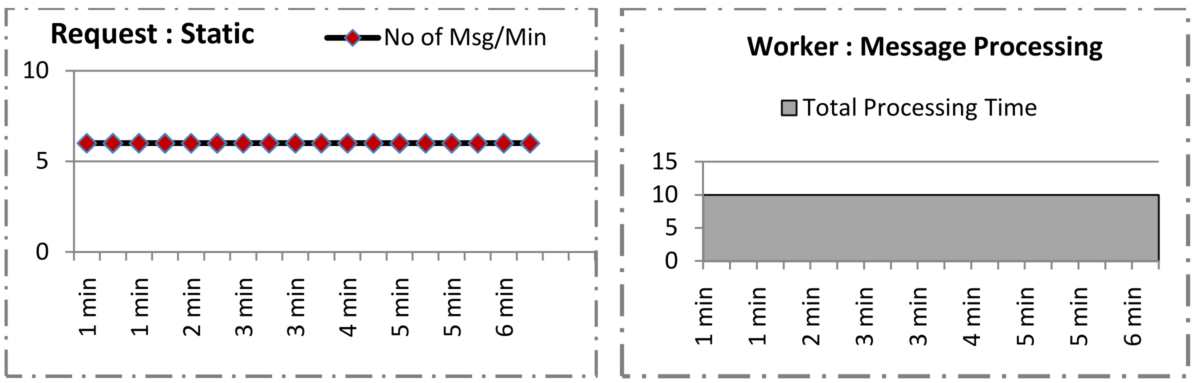


Figure 5.1.1: Static Load generated by the Requester & Message processing duration by the Worker

Note : For every figure horizontal axis represent the execution duraion (min).

5.1.2 Test case 2 : Static Increase

In this test case, only one Requester sends message to the Amazon SQS queue in increasing manner. In each time slot, additional messages are coming to the queue that increases the Worker load for processing a message. Total execution duration was 5 minutes. In the first minute of execution the rate of msg/min is 4, in next minute 6, in 3rd minute 12msg/min, then 15/min and in last minutes 20/min.

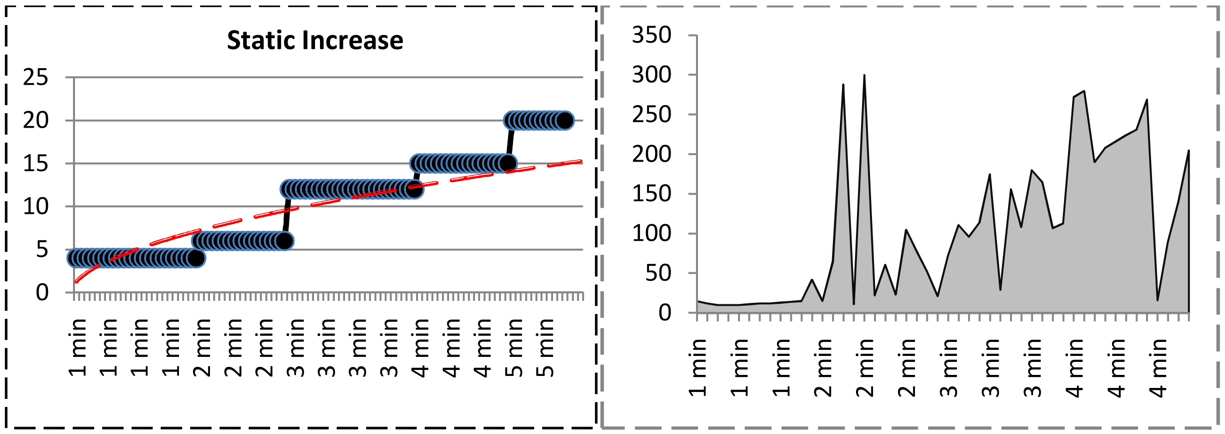


Figure 5.1.2: Static Increasing Load by Requester & processing duration by one Worker

The figure 5.1.2 represents the worker processing time during the test case execution period. After monitoring, it was found that at the beginning worker takes nearly same processing time and behaves like as FIFO while reading read messages from the Queue. However, the rapid increase of incoming messages to the queue decreases the chance of reading message as FIFO. Rather, it takes the message from the queue in arbitrary order and needs more time to process. This shows the arbitrary message reading policy of Amazon SQS queue.

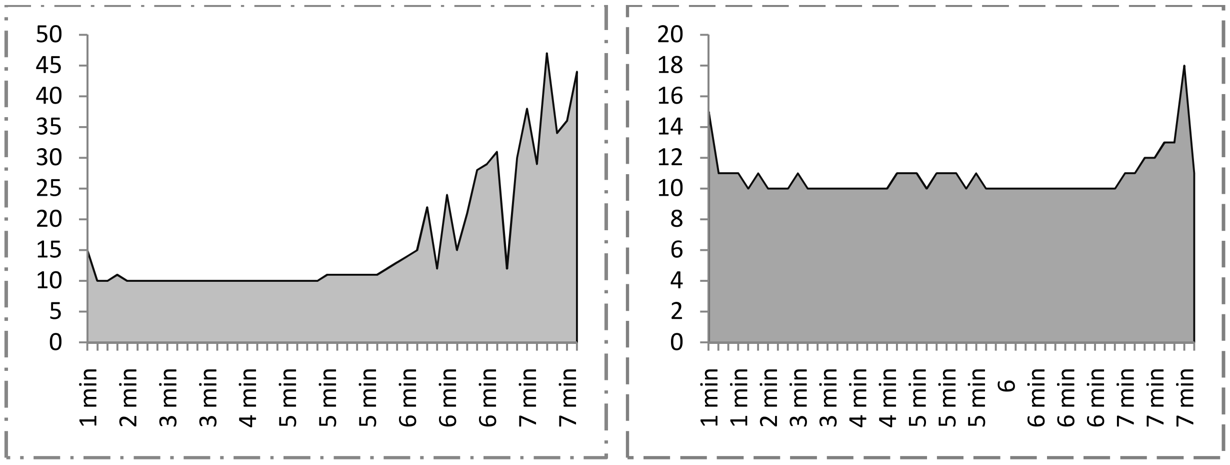


Figure 5.1.3: Message processing duration by two and Three Workers respectively

In order to reduce worker load (message processing time), number of worker was increased. As seen in figure 5.1.3 two workers are reading message simultaneously from the queue which results less time to read message.

After adding one more Worker, in total three are running simutinously, takes approximately constant processing time during the entire execution period.

5.1.3 Test case 3 : Sudden Increase

As seen in figure 5.1.4., at one particular time, Requester increases the number of message sending to the queue dynamically and makes the Queue full of message. It ultimately generate huge overload to the Worker while processing a message.

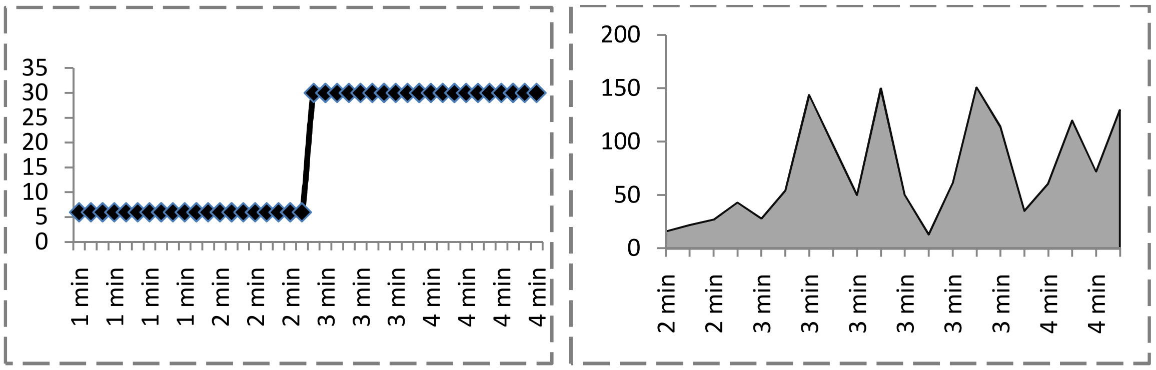


Figure 5.1.4: Sudden Load Increase by Requester and message processing by Worker

In this scenario it has been found that, at the beginning worker behaves as FIFO. However when the queue is overloaded with messaged then it follows the random method to read message from the queue that results higher processing time (10 times in few cases).

5.1.4 Test case 4 : Peak Increase

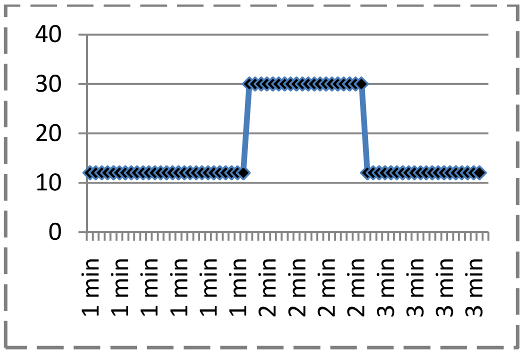


Figure 5.1.5: Peak Load increase by the Requester

In this test case figure 5.1.5, Requester send message to the queue in typical rate, but at particular time slot it increases the message rate dynamically and continues for certain duration. After a while it comes back to previous message rate.

The test observation in figure 5.1.6 shows that the workload was distributed among multiple workers. Typically one worker takes more than ten times higher time than two workers to process one message. When two workers are running simultaneously, after the peak increase (at first minute) both worker comes peak to normal behavior. Similar to the others scenario, chance of behaving as FIFO in message reading reduces as soon as the queue is overloaded with messages.

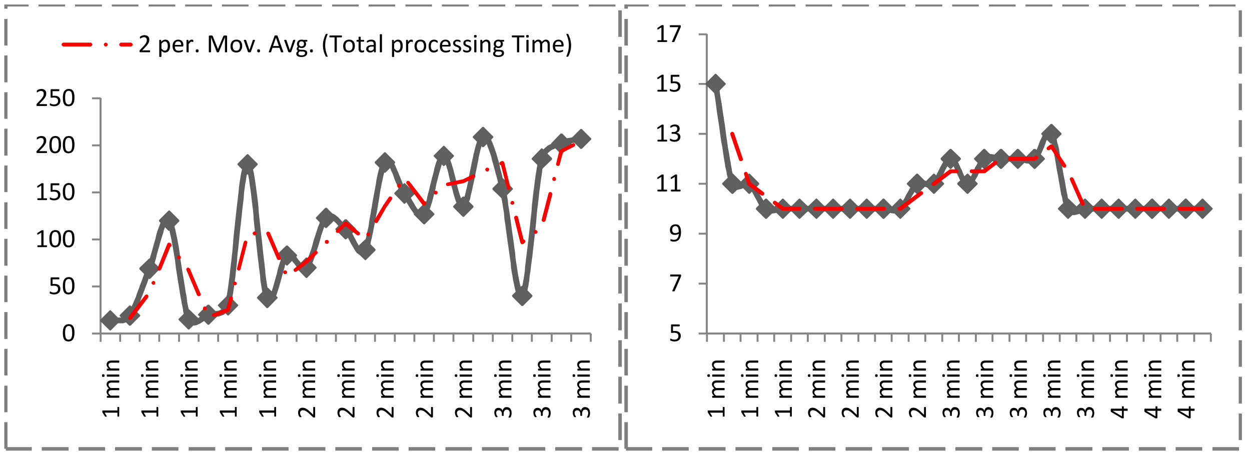


Figure 5.1.6: Message processing time by one Worker and two Workers respectively

5.1.5 Test case 5 : Static Decrease

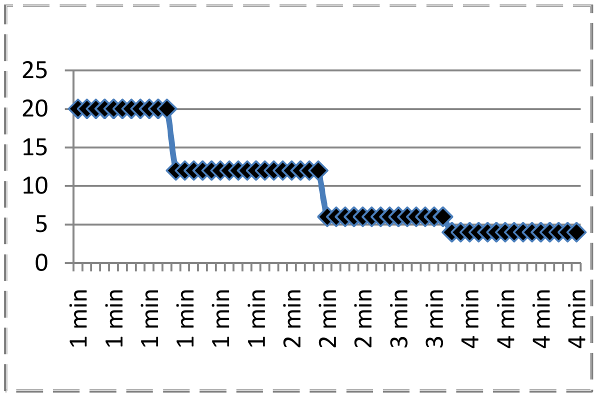


Figure 5.1.7: Static Load decrease by the Requester

As seen in figure 5.1.7 requester sends message to the queue in decreasing manner. As the execution time progresses, number of messages coming to the queue decreases.

In the middle of the execution such as minute 2-3 when the queue is full with messages, processing time increases rapidly. After adding one more worker, processing time becomes to one fourth and nearly constant at the end of execution period. Though this provides better performance to reach deadline, besides it indicates over provisioning of resources.

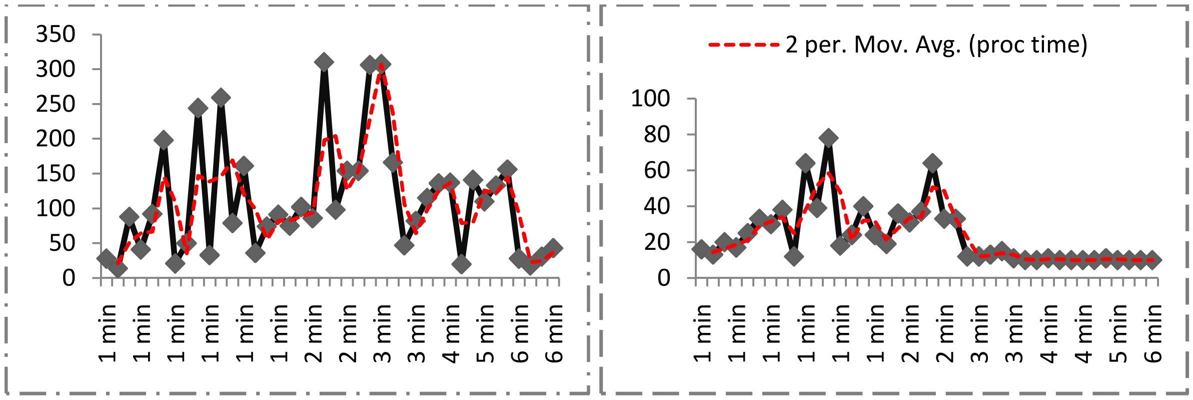


Figure 5.1.8: Message processing time by one Worker and two worker repectively

5.1.6 Sudden Decrease

In figure 5.1.9 we can observe that at one point in time, Requester decreases the number of message sending to the queue indolently which supposed to reduce the worker message time and trends to move to the balanced processing time. However, as at the beginning of execution the queue is overloaded, the decreased message rate is not able create an impact to the processing time.

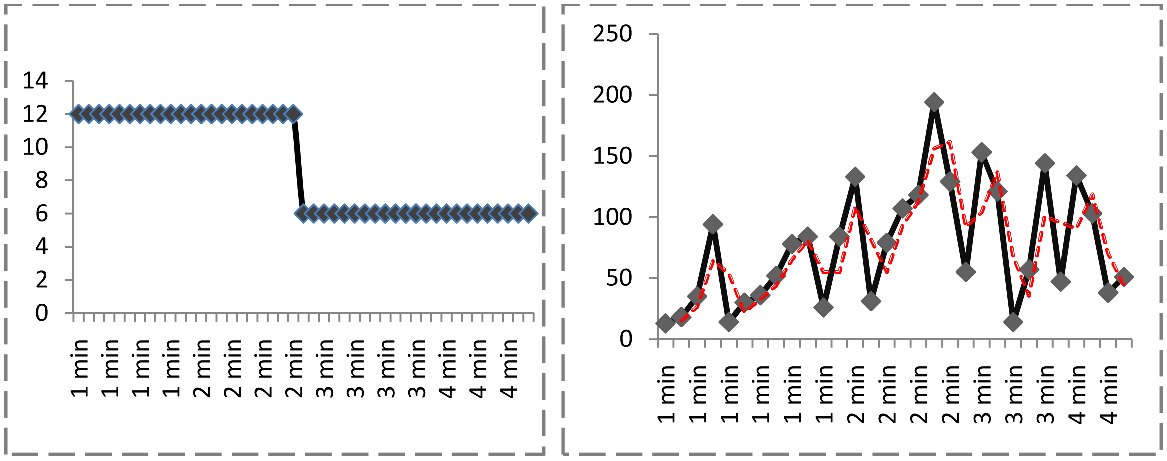


Figure 5.1.9 Sudden Load decrease by the Requester and message proceeing by the Worker.

5.1.7 Peak Drop

In the last test case seen in figure 5.1.10, in a particular time slot Requester decreases the number of message sending to the queue indolently. After a while that it again comes back to previous message sending rate. So, the queue must have fewer messages during that period of particular time slot.

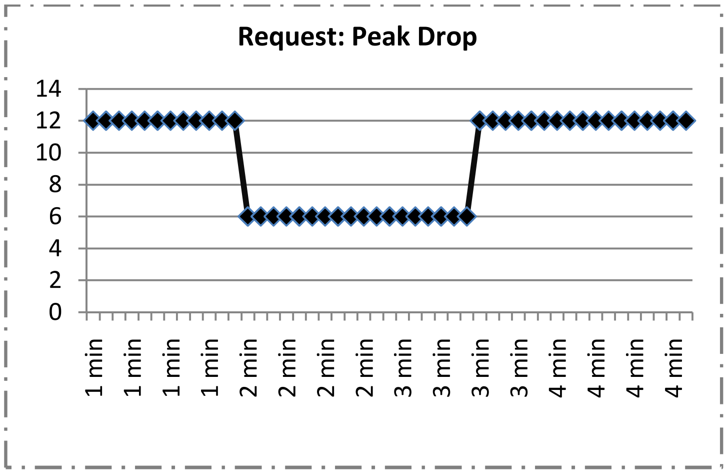


Figure 5.1.10 Peak Load drop by the Requester

In figure 5.1.11, message processing time by the worker should reduce eventually. However, the test observation says that since at the beginning of the execution queue was overloaded, the decreased message rate even could reduce the processing time. Rather, adding one more worker provides better result and brings down the processing time to one fifth of previous execution. This eventually shows the randomized message reading behavior of Amazon SQS Queues.

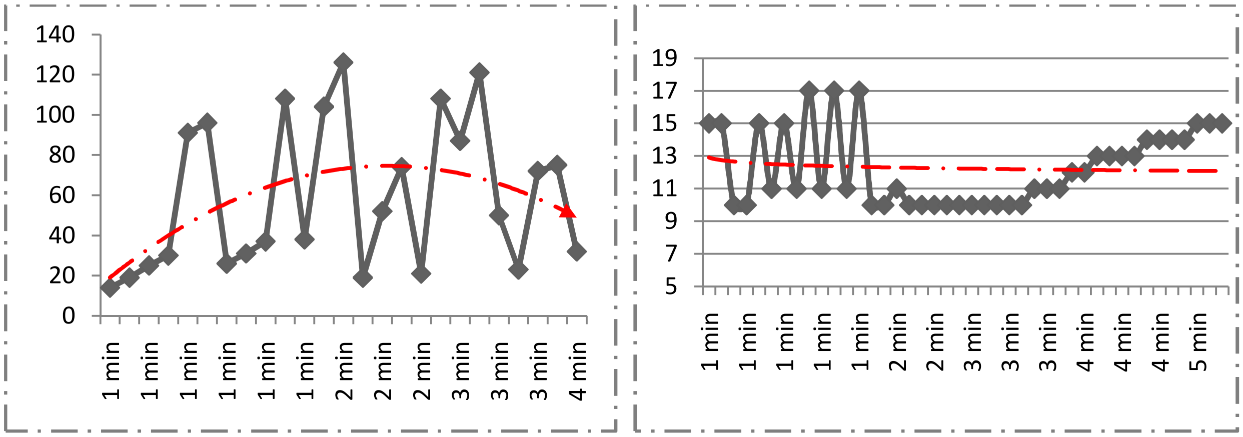


Figure 5.1.11: Message processing time by one Worker and two worker repectively

6. Summary

This thesis has proposed an optimal resource provision method for the Cloud service providers that can be integrated into their Cloud. Cloud provider will be benefited by optimizing the energy consumption and their maintenance cost, maximizing the Cloud resource utilization, increasing the system throughput and ensuring a cost-efficient on demand resource provisioning into the Cloud. On the other hand, Cloud application users can get the price efficient, deadline oriented and flexible application deployment facility into the Cloud with this event based application management system. The proposed optimized queue scaling mechanism reduces the total power consumption and operational cost compared to current infrastructure offerings from the Cloud vendors.

Though every algorithm has its particular benefits and disadvantages, this thesis enables Cloud computing users to allocate resources dynamically within their limited budget and time constraint. It is quite noticeable that typical static resource provisioning [2] in traditional data center scenario wastes energy and money while providing minor increases in availability. This thesis enables the elasticity rules to the application provider to expand and shrink their resources as per demand. It has also provided a competent capacity planning model to the Cloud provider for provisioning the resources. The energy optimization theory will significantly reduce the amount of power consumption that enables to maintain the concern of Green Computing. The Cost function algorithm and policy based algorithm ensures the maximum resource utilization within minimal cost and energy. It is also capable to meet the deadline of application provisioning into the cloud.

The depicted system architecture describes each of the components such as monitoring system, optimizer and provisioning manager that need to work simultaneously to provision Cloud resources. It is able to allocate required resources as per the trigger sent to the provisioning manager.

A prototype was implemented using AWS Eclipse toolkit and AMAZON Elastic Cloud EC2 environment. AMAZON S3 storage system was used to store a configuration file which can be used to keep the rules and policies of SLAs in the production environment.

6.1 Limitation & Outlook

Cloud computing is still a relatively new paradigm, so the changes in standards, infrastructural capabilities and component APIs are inevitable. Among other major challenges during the implementation period, the customized pricing and energy consumption information into the Cloud environment was highly mentionable. Unfortunately, due to the limitation of the used Cloud properties, it was not possible to collect this information. The proposed scaling mechanism in this thesis would be more effective if following issues can be solved.

- **Implementation inside data center environment**

For designing the optimize solutions of scheduling and resource provisioning of applications inside Cloud data center, factors such as cooling, network, memory and CPU need to be considered. Though the consolidation of VMs is effective to minimize the overall power usage of data center, it raises the issues of redundancy and geo-diversity [73] to maintain the SLAs with users.

- **Lack of enhanced cost model**

Due to the lack of running resource costs information such as traffic or provision cost, the cost function became quite simple. However, it can be integrated with the addition information.

- **Challenge to Green Computing**

To ensure the Green Computing Cloud provider needs to measure the existing power and cooling design, power consumption of services and their cooling requirements. To do that they should have appropriate modeling tools to measure the energy usage of all the components and services of from a user PC to data center where the application is hosted.

7. Bibliography

7.1 Glossary

Access Key ID	An alphanumeric token that uniquely identifies request sender. This ID is associated with your Secret Access Key.
Amazon EC2	The Amazon Elastic Compute Cloud (Amazon EC2) is a web service that enables user to launch and manage server instances in Amazon's data centers using APIs and available tools and utilities.
AMI	An Amazon Machine Image (AMI) is an encrypted machine image stored in Amazon Simple Storage Service (Amazon S3). It contains all the information necessary to boot instances of your software.
Amazon SQS	Amazon Simple Queue Service (Amazon SQS) offers a reliable, highly scalable, hosted queue for storing messages as they travel between computers.
Availability Zone	Amazon EC2 locations are composed of Regions and Availability Zones. Availability Zones are distinct locations that are engineered to be insulated from failures in other Availability Zones and provide inexpensive, low-latency network connectivity to other Availability Zones in the same Region.
Auto Scaling group	A representation of an application running on multiple Amazon Elastic Compute Cloud (EC2) instances.
AWS	Amazon Web services
EC2 instance	A virtual computer running in the cloud. EC2 instances are launched from an Amazon Machine Image (AMI).
Life cycle	Auto Scaling term that refers to the life cycle state of the EC2 instances contained in an Auto Scaling group. EC2 instances progress through several states like as Pending, In Service, Terminating and Terminated over their lifespan.
LoadBalancer	Elastic Load Balancing key term. A LoadBalancer is represented by a DNS name and provides the single destination to which all requests intended for your application should be directed.

7.2 References

1. Hellerstein, J. ; Diao, Y. : *Feedback control of Computing System.* Wiley Online Library, 2004 (Cited on pages 3 & 5)
2. Vaquero, Luis M. ; Buyya, R. : *Dynamic Scaling Application in Cloud.* ACM SIGCOMM Computer Communication Review, 2011 *(*Cited on Page 47*)*
3. Leymann, F. ; Mietzner, R.: Applications in the Cloud. ITPC Cloud Day, 2009 (Cited on page 11)
4. Fehling, C : Provisioning of Software as a Service Applications in the Cloud, 2009 (Cited on page 14)
5. Leymann, F. and Fritsch, D: *Cloud Computing: The Next Revolution in IT.* Proceedings of the 52th Photogrammetric Week, *2009. (Cited on page 5)*
6. Nowak, A. and Leymann, F. and Mietzner, R.: *Towards green business process reengineering*. Service-Oriented Computing, 2011 (Cited on page 5)
7. Lim, H. C. ; Babu, S. : *Automated Control in Cloud Computing Challenges and Opportunities. 2009* (Cited on page 2)
8. Parekh, S. : *Using Control Theory to Achieve Service Level Objectives In Performance Management*, October 23, 2000 (Cited on page 3)
9. Chen, Y.; Tsai, S.: *Optimal Provisioning of Resource in a Cloud Service*, 2007. (Cited on page 7)
10. Hohpe, G. ; Woolf, B.: *Enterprise integration patterns: Designing, building, and deploying messaging solutions*. Addison-Wesley, 2004 (Cited on on page 38)
11. Zhu, J. : *Dynamic Provisioning Modeling for Virtualized Multi-tier Applications in Cloud Data Center* . 2010 IEEE 3rd International Conference on Cloud Computing, 2010 (Cited on pages 373-375)
12. Buyya, R.: *Cloud Computing Principles and Paradigms*. Wiley , 2011 (Cited on pages 130-136)
13. Ogata, Katsuhiko*; Modern Control Engineering*. Prentice Hall, 3rd edition, 1997 (Cited on page 76-78).

14. Parekh, S. : *Using control theory to achieve service level objectives in performance management.* Springer, 2002 (Cited on page 7)
15. Allspaw, J.: *The Art of Capasity Planning,* O'REILLY, 2008 (Cited pages 19,26 & 122)
16. AMAZN.COM : *AWS Simple Queue Services.* - http://aws.amazon.com/sqs/ (Cited on page 2)
17. AMAZN.COM : *AWS Simple Queue Services Visibility Timeout.* - http://docs.amazonwebservices.com/AWSSimpleQueueService/latest/SQSDeveloperGuide/ (Cited on page 2)
18. Mao, Ming; Li,Jie; Humphrey, Marty; *Cloud Auto-scaling with Deadline and Budget Constraints.* ACM/IEEE International Conference on Grid Computing , 2010 (Cited on Page 2)
19. SHLOMO Swidlers : *EC2 Life Cycle .* - http://shlomoswidler.com/2009/07/ec2-instance-life-cycle.html (Cited on Page 1)
20. AMAZN.COM : *Amazon Elastic Cloud EC2.* - http://aws.amazon.com/ec2/ (Cited on Page 3)
21. Manber, U.: *Introduction to algorithms: a creative approach*. Addison-Wesley, 1989 (Cited on pages 36 &48)
22. HHS.COM : *Capacity Planning*. - http://www.hhs.gov/ocio/eplc/EPLC%20Archive%20Documents/34-Capacity%20Planning/eplc_capacity_planning_practices_guide.pdf (Cited on Page 5)
23. KINGSTON.COM : *Capacity Planning* .- http://www.kingston.com/branded/pdf_files/handrscapacity_wp.pdf (Cited on Page 4)
24. Teamquest.Com : *Capacity Planning*.- http://www.teamquest.com/pdfs/whitepaper/tqwp23.pdf (Cited on Page 5)

25. Erl, T.: *Service-Oriented Architecture: Concepts, Technology, and Design*. Prentice-Hall, 2005 (Cited on page 19)

26. CLOUDTWEAKS.COM : *Cloud Platform* .- http://www.cloudtweaks.com/2011/05/auto-scaling-strategies-for-windows-azure-amazons-ec2-and-other-cloud-platforms/
27. MSDN.COM : *Windows Azure.*- http://blogs.msdn.com/b/gonzalorc/archive/2010/02/07/auto-scaling-in-azure.aspx (Cited on page 5)
28. AMAZN.COM *: Auto Scaling Developer Guide*.- http://aws.amazon.com/as-dg/
29. Azeez, Afkham; *Auto-scaling Web Services on Amazon EC2*, University of Moratuwa, 2009 (Cited on page 41)
30. AMAZN.COM: *Autoscaling*.- http://aws.amazon.com/autoscaling/ (Cited on page 3)
31. Cloud Compuning Journenal : *AutoScaling in Windows Azure*.- http://srinivasansundararajan.sys-con.com/node/1626508/mobile (Cited on page 9)
32. AMAZN.COM: *Elastic Load Balancing.*- http://aws.amazon.com/awseb-ug/
33. SCALR.NET : *Architecture.*- http://wiki.scalr.net/Getting_Started (Cited on Page 5)
34. TECHCRUNCH.COM : *Scalr auto Scaling.*- http://techcrunch.com/2008/04/03/scalr-the-auto-scaling-open-source-amazon-ec2-effort/
35. MSDN.COM : *Windows Azure.*- http://archive.msdn.microsoft.com/azurescale/Release/ProjectReleases.aspx?ReleaseId=4167 (Cited on page 11)
36. MSDN.COM : *Windows Azure.*- http://blogs.msdn.com/b/gonzalorc/archive/2010/02/07/auto-scaling-in-azure.aspx
37. Cloudcomputing.sys-con.com *: Dynamic Scaling and Elasticity - Windows Azure vs Amazon EC2*.- http://cloudcomputing.sys-con.com/node/1626508 (Cited on page 2)
38. Paraleap.com : *AzureWatch*.- http://www.paraleap.com/(Cited on page 3)
39. RIGHTSCALE.COM: *Social Networking*.- http://st.free-lance.ru/projects/upload/f_4be861b83ddf9.pdf

40. RIGHTSCALE.COM: *Scalable Website.-* http://www.RIGHTSCALE.COM/products/cloud-computing-uses/scalable-website.php (Cited on page 3)
41. RIGHTSCALE.COM: *Dev & Test White Paper.-* http://www.RIGHTSCALE.COM/info_center/white-papers/RightScale-Development-and-Test-White-Paper.pdf (Cited on pages 6 & 9)
42. RIGHTSCALE.COM: *Test Benefits.-* http://www.RIGHTSCALE.COM/products/plans-pricing/dev-test-features-benefits.php
43. RIGHTSCALE.COM: *Dev & Test White Paper.-* http://www.RIGHTSCALE.COM/info_center/white-papers/RightScale-Quantifying-The-Benefits.pdf (Cited on page 3)
44. AMAZN.COM: *S3 Storage*.- http://aws.amazon.com/s3/
45. RIGHTSCALE.COM: *Grid Business.-* http://www.RIGHTSCALE.COM/info_center/white-papers/Grid-Whitepaper-Business.pdf
46. Britannica.com: *Event in Automation .-* http://www.britannica.com/EBchecked/topic/44836/automata-theory/21502/Input-events-that-affect-an-automaton (Cited on page 7)
47. RIGHTSCALE.COM: *Plans & Pricing.-* http://www.RIGHTSCALE.COM/products/plans-pricing/grid-edition.php
48. MICROSOFT.COM: *Virtual Network.-* http://www.microsoft.com/windowsazure/features/virtualnetwork/
49. S. Hazelhurst; *Scientific computing using virtual high-performance computing: a case study using the Amazon elastic computing cloud, in: Proceedings of the 2008 annual research conference of the South African Institute of Computer Scientists and Information Technologists on IT research in developing countries: riding the wave of technology, ACM New York, NY, USA, 2008, pp. 94–103.*
50. AMAZN.COM: *AWS Elastic Beanstalk.-* http://aws.amazon.com/elasticbeanstalk/ (Cited on page 3)
51. Cloudcomputing.sys-con.com :*Top 30 Cloud Service Providers.-* http://cloudcomputing.sys-con.com/node/1513491 (Cited on page 1)

52. VMWARE.COM :*Virtualization & Virtual Machine .-* http://www.vmware.com/virtualization/virtual-machine.html (Cited on page 1)
53. SoftSummit.com :*Utility based pricing model.-* http://www.softsummit.com/library/presentations/2003/HoganS.pdf
54. Ludwig, H.: *A service level agreement language for dynamic electronic services.* Electronic Commerce Research, 2004 (Cited on page 2)
55. ORACLE.COM : *Trigger.-* http://download.oracle.com/docs/cd/B19306_01/server.102/b14220/triggers.htm
56. AMAZN.COM: *LoadBalancing.-* http://aws.amazon.com/elasticloadbalancing/ (Cited on page 4)
57. AMAZN.COM: *Amazon Machine Images*.- http://aws.amazon.com/amis (Cited on Page 2)
58. SCALR.NET : *API*.- http://wiki.scalr.net/API
59. AMAZN.COM: *EC Instance.-* http://aws.amazon.com/ec2/instance-types/
60. AMAZN.COM: *Amazon CloudWatch*.- http://aws.amazon.com/cloudwatch/ (Cited on page 5)
61. Helpcenter.com : *NameSevers*.- http://help.godaddy.com/article/45
62. Berl, A., Gelenb, E. : *Energy-Efficient Cloud Computing,* 28 July 2009.
63. MSDN.COM : *3-Tier Architechture*.- http://msdn.microsoft.com/en-us/library/ms685068(v=vs.85).aspx (Cited Page 2)
64. TPUB.COM : *Control System .-* http://www.tpub.com/content/neets/14187/css/14187_92.htm (Cited on page 1)
65. AMAZN.COM: *White paper on cloud architectures* http://aws.typepad.com/aws/2008/07/white-paper-on.html
66. FORMDUAL.CH: *Round Robin Database*.- http://www.fromdual.ch/round-robin-database-storage-engine (Cited page 2)
67. Lipari, G.: *Earliest Deadline First.* Scuola Superiore Sant'Anna, Italy, 2005 (Cited on page 3)
68. AMAZN.COM: *Requester-Worker Paradigm.-* http://docs.amazonwebservices.com/AWSMechTurk/2008-02-

14/AWSMechanicalTurkRequester/Concepts_RequestersAndWorkersArticle.html (Cited on Page 1)

69. Dougherty, B., Whiteb, J.: *Model-driven Auto-scaling of Green Cloud ComputingInfrastructure.* Science of Computer Programming, 2010 (Cited on Page 5, 7 & 9)
70. Kupferman, J., Silverman, J.: *Scaling Into the Cloud.* CS270 Advanced Operating Systems. 2011 (Cited on page 2)
71. Abrahiem, R.: *A new generation of middleware solutions for a near-real-time data warehousing architecture.* IEEE International Conference, 2007 (Cited on page 3)
72. Nauturenet.com : *Geo-Diversity* :- http://www.naturenet.net/biodiversity/geodiversity.html (Cited on page 1)

All Links were last followed on August 20, 2011.

Declaration

All the work contained within this thesis, except where otherwise acknowledged, was solely the effort of the author. At no stage was any collaboration entered into with any other party.

Sams Ul Arefin