Autonomous Virtual Machines for Ensuring QoS of Distributed Web Services Using Evolutionary Game Theory

A proof of concept of self-organising virtual machines

A.S.M. Samiul Saki Chowdhury

Thesis submitted for the degree of
Master in Network and System Administration
30 credits

Department of Informatics
Faculty of Mathematics and Natural Sciences

UNIVERSITY OF OSLO

Spring 2018

Autonomous Virtual Machines for Ensuring QoS of Distributed Web Services Using Evolutionary Game Theory

A proof of concept of
self-organising virtual machines

A.S.M. Samiul Saki Chowdhury

Autonomous Virtual Machines for Ensuring QoS of Distributed Web Services Using Evolutionary Game Theory

http://www.duo.uio.no/

Printed: Reprosentralen, University of Oslo

Abstract

The virtualisation of hardware resources is one of the universal features of modern-day cloud computing. The agility and effectiveness of hardware resource pool virtualisation, geographic diversity and universal connectivity of various inter-connected components play a vital role in cloud computing. The research of this field is now magnified towards the study and development of secure and stable migration of the shared virtual systems. Efficient usage of physical components by sharing over a public space is now one of the main focus of the traditional and upcoming cloud providers and the enterprise networks.

It is already in practice to clone virtual machines (VMs) in order to create a more scalable and flexible infrastructure over the internet. Popular technique such as migration of VMs while maintaining the optimal quality of service (QoS) of the distributed web services across the globe has also become an on-demand feature of cloud infrastructure. Administration of these virtual systems with various types of services are now needed to be more flexible towards adapting to the clients' requests. The broad demand for these services in large-scale requires some sort of autonomy in the system as it is becoming more complicated to maintain them manually. The other feasible options, that are mostly adopted by data centres, are to implement a centralised system which controls all the VMs in the cloud. This solution is adequate until the backbone of the central system breaks down or cannot recover from a node depletion. The concept of a self-aware VM is to perform tasks independently, self-regulate its behaviour to maintain stability by analysing the network condition and other external/internal attributes in order to introduce an equivalence in the cloud architecture. This phenomenon of virtualisation demands an effective algorithm that can conduct and administer the emergence behaviour of the virtualised systems. This designed algorithm need to approach an autonomy with self-governance and self-maintenance feature in the network to maintain system-wide stability.

In this project, we propose different variants of migration algorithms by adopting the evolutionary game theory, to achieve the emergence behaviour of the VMs in a network by equalising the average response times of the entire connected network. Our algorithms are applied to the VMs in the network, to make them be more independent and autonomous. Using these algorithms each individual VM can observe and learn the network condition at a particular moment, implement the algorithm, analyse and select the affected data-centres as the migrating destination, whilst maintaining the quality of service of the distributed web services. These algorithms are designed to stabilise the average response time in all the adjoined data-centres in a network by distributing the idle migrating VMs among each other without any requirement of a centralised management.

Furthermore, we perform multiple experiments by implementing these algorithms in simulations and in real-life cloud infrastructure test-bed and accumulate results of our design performance of modified self-managed and self-coordinated VMs. We analyse and discuss the performance of the algorithms in VM instances deployed in the cloud platform. The results show, using our migration algorithms, we can equalise the response time in the entire network with the self-organised VMs, by applying an autonomy in the system, while providing a satisfactory QoS of distributed web service

Preface

Entrance enthusiasm and revelation have always emerged as a fundamental part in the accomplishment of any task. This thesis is the result of the NSA5930 Master's thesis project, which is corresponding to 30 ECTS points, at the masters programme Network and System Administration (NSA), Department of Informatics, Faculty of Mathematics and Natural Sciences, University of Oslo (UiO) in Oslo, Norway. The work has started from January 5, 2018 and ended on May 22, 2018.

I would like to thank my thesis supervisor, Associate Professor Anis Yazidi, Oslo Metropolitan University (OsloMet), for his encouragement, ideas, guidance and support from the initial to the final level throughout the entire project. With his supervision I have learned a great deal about design principal behind cloud computing the technical implementations of scientific concepts.

I am also thankful to Associate Professor Hårek Haugerud for all his help and guidance on setting up the cloud infrastructure in order to implement and achieve the project goal. I would like to thank University of Oslo and to Oslo Metropolitan University for offering me this masters programme. I am grateful to my wife Kari-Anette Nilsen for her inspiration during this master thesis period and her devoted support.

A.S.M. Samiul Saki Chowdhury

Oslo
May 20, 2018

Contents

List of Figures

List of Tables

List of Acronyms

- **DNS** Domain Name System
- **EGT** Evolutionary game theory
- **EWMA** Exponential Weighted Moving Average
- **GCP** Google Cloud Platform
- **I/O** Input/Output
- **IT** Information Technology
- **KVM** Kernel-Based Virtual Machine
- **OS** Operating System
- **QoS** Quality of Service
- **RMS** Resource Management System
- **SLA** Service Level Agreement
- **VM** Virtual Machine

Part I

Introduction

Chapter 1

Introduction

The modern-day information technology (IT) is developed and moderated continuously over era to this point of present-day. Computer components and their inter-communication algorithms are being improved to adapt to the latest technological demand. In the course of time, the mathematical operations in electronic hardware, were formalised and became more understandable and familiarise well enough to be stated formally and proven with complex algorithms followed by scaling the system to a giant network of computers and smart devices. The evolution in IT industry is mainly evaluated by the emergence and development of communication technology. The suitable and affordable cost of smart devices around the globe creates the requirement of extensively active and highly effective connectivity between those devices. Resources in a system are not bound to be kept in one single machine but should be distributed and shared among various clients/users. Data are constantly in need to be shared over the different form of components. This high demand for big data computation requires a higher quality of network speed as well as space to keep a large amount of data or information. A client, not only can access the resources over the internet but also can use the remote machine as it is handled locally. Present day communication technology is in need of a productive way to share these resources which requires a efficient management of system components.

1.1 Motivation

With the early age mainframe computing, multiple users were capable of accessing a central computer through simplified terminals, whose only function was to provide access to those mainframes. It was impractical for an organisation to buy and maintain one mainframe computers for every employee, mostly due to the reason the costs to buy those computers. Nor did the typical user need the large (at the time) storage capacity and processing power, that a mainframe provided. Providing shared access to a single resource was the solution that made economic sense for this sophisticated piece of technology. The idea of virtual machines (VMs) first appear in the late 1960s and continue to arise on active development when the resource pool sharing start becoming more expensive due to the vast need of user-demand. The limitation of hardware components brought out the motivation behind running multiple operating systems on a single machine. The concept was first introduced by allowing time-sharing among several single-tasking operating systems. The implementation of time sharing by different users was first introduced by IBM [1]. But unlike virtual memory, a system VM entitle users to write privileged instructions in the code which gave the advantage to add input/output (I/O) devices that are not allowed by the standard systems. As technology improved, newer systems became capable of managing memory sharing among multiple VMs on the same computer operating system (OS) or on different OS over the internet. Slowly It became possible to execute one or more operating systems simultaneously in an isolated environment by using virtualisation software like VMware. Multiple VMs running their own guest OS are frequently engaged for server consolidation.

The server is the concept when a pool of hardware resources are organised and maintained in order to let a group or remote clients connect with them and share the same pool to spawn individual VMs [2]. This makes the most uses of the resources as it is considered to be the most innovative way to cut down product costs. Virtualisation came to drive the technology and was an important catalyst in the communication and information revolution.

The concept of cloud computing indicates the VMs that are connected globally over the internet in different servers despite the location. Cloud capability often refers to the hardware resource capabilities of the servers. The cloud is the delivery of on-demand computing resources, everything from applications to data centres. Cloud providers are the organisations which typically use a payment model to let the client borrow their components. It allows companies to avoid or minimise up-front IT infrastructure costs. The manufacturing and maintenance cost of a product is one of the main purposes of sharing resources between different clients. The main idea of sharing is the efficient use of all the resources between various systems. If a user needs to work on a project where a highly efficient system or higher quality hardware are required, it is not necessary for that user to buy a new product for that particular purpose only, rather the user can easily lend/borrow a resource from another client over the internet with some condition applied. Cloud computing in IT paradigm enables ubiquitous access to shared pools of configurable system resources and higher level services that can rapidly be provisioned with minimal management effort, often over the internet. This type of computing depends on the sharing of resources to achieve economy of scale and coherence.

The services offered by vendors and other developers are versatile and increasing globally as businesses. As the network of the big shared system grows, the number of components relies on running those services are growing as well. In a large-scale network, complex algorithms and process-hungry services are constantly requiring the support of larger amount hardware components. The resource pool to run the deployed services is now shared between end-users. The idea of sharing components, for the specific purpose is becoming more popular which needs an efficient resource management system. This opens up the opportunities of the vast field of development on VM management by implementing resilient algorithms on the computing systems. Not only it is required to spawn VMs on the server but also administration, performance, scaling and migration of those VMs are becoming present-day research interest. The goal is to optimise the performance of a VM to response and perform according to multiple users request to the different part of the world. Moreover, the term VM migration is the task of moving VMs with services, from one physical hardware environment to another, preferably without losing its state. As the application of deploying virtual services grows, interest on the development of sharing VMs over the internet became more popular. The development of sharing client's to data/services among each other more efficiently accelerated the research on better management of VMs in a infrastructure for the appropriate use of resources and improvement on reducing service depletion due to poor error handling and management of the VMs. At this point, when every system is trending to complex systems and utilisation of big data becoming a colossal problem, it is a concern of legitimate VM administration in cloud computing. The majority of the services deployed in the cloud and it is not possible to control the administration manually anymore. An autonomic system is essential for this sort of management to avoid human probed error as the minimum. This means suitable algorithms must be implemented to control and manage VMs to maintain the quality of services (QoS), migrate the services, optimise the performance and preferably in a distributed manner rather than centralised fashion.

In this project, we advocate a system design which can inspire future researchers to achieve an improved working solution for autonomous VM migration to provide QoS of distributed web services over diverse hardware platforms.

1.2 Problem Statement

For migration of virtualised systems, administrators move logical hardware pieces, between physical servers or other hardware pieces that do not have a physical shell or composition. Modern-day services often provide migration functionality to make it easier to move VMs without doing a lot of other administrative work [3]. To ensure QoS on those offered distributed services faces various objectives that must be taken into account whilst deploying. Keeping the downtime minimum or saving the state of machines when migrating, tends to be a tedious and resource greedy process. An algorithm based on such as evolutionary game theory, needed to be designed which can successfully be integrated into each VM. The algorithm can be adapted from nature as an example such as animal behaviour in collective. This will make VMs more autonomous and function in a distributed manner, i.e., no centralised governing is needed. The algorithm should define what will be VMs main principle for moving into a different physical location. Its also need to be recognised that this particular algorithm should concentrate on how to make VMs be more efficient when it comes to keeping the state of the services. Furthermore, the performance of system design must be evaluated by putting it to the tests in various test-bed scenarios and measure, analyse and compare the experimental results extracted from the systems to assess system design.

The problem arises when the lack of managing algorithm to automate VM management fails to process the clients request incoming to services. In this masters thesis, we are addressing these problems for VM automation:

> ***"How to design a suitable algorithm which can automate the migration demeanour of VMs in order to adopt evolutionary game theory and equalise the average response time of different regions? How to achieve autonomous characteristics in the VMs while maintaining the QoS of the distributed web services without the assistance of centralised systems?"***

1.3 Report Outline

Rest of our thesis report is organised as follows:

- In Chapter 2.6, the enabling technologies that are implemented in this project have been discussed. An overview of the tools that are used in the system design are also provided and some of the interesting related work that had been done on VM migration, self-awareness of VMs and algorithm based on evolutionary game theory are reviewed.
- In Chapter 3, the methodology of adapting and evaluating the implemented algorithms using evolutionary game theory with different variants are thoroughly explained.
- In Chapter 4, results of the test simulations are analysed with numerical results and graphs. Implementation of proposed algorithm in real-life test-bed scenarios have been conducted and numerical data has been gathered and analysed in this chapter.
- In Chapter 5, we discuss the performance and constraints of the proposed evolutionary algorithm.
- Finally, in Chapter 6, the conclusions and future work are presented.

Part II

The Project

Chapter 2

Enabling Technologies and Related Work

A designer of cloud infrastructure has multiple choices of protocol and standards, ranging from the simple to the amazingly complex design. In hardware virtualisation, physical hardware pieces are carved up into a cluster of VMs, logical hardware pieces that do not have a physical shell or composition, which are essentially just programmed pieces of an overall hardware system. In a virtualisation setup, typically, a central hypervisor, allocates resources like CPU and memory to VMs. For instance, in older networks, most of the individual elements were physical workstations, such as desktop PCs, which were connected by Ethernet cables or other physical connections. By contrast, VMs do not have a physical interface. They do not have a box or shell or anything to be mobile. But they can be connected to the same keyboards, monitors and peripherals that humans have always used to interact with personal computers. As discussed in Chapter. 1, the technology for maintaining cloud computing has emerged from a long history of development on VM sharing. These stock VMs are quite field ready to be deployed in a cloud for an autonomous network. It is required to be a simple yet competent configuration of the VMs to sense and react to the network it is connected to or the services it is running. Furthermore, these The legacy systems usually have a controller that maintains the connectivity and method to respond to different network situations. These controllers can be referred to be as the brains of the system where the VMs that provide the services can be referred to be as the worker systems. Having an autonomous system provides the VMs to avoid the administrative boundaries, by taking control of making their own routing and functional execution on its own. The design of such self-aware systems necessitates optimised cloud network setup along with other attributes. The enabling technology of such configuration requires various cloud computing tools to be operated in a consensus manner. The following sections briefly describe some of the most substantial tools have been acclimated in this project.

2.1 Hypervisor

Cloud computing entails clients to be able to access a VM and also use those machines anywhere. A hypervisor manages these VMs. It is a program that enables servers/users to host several different VMs on a single hardware. On each of this machine or operating system (OS), an individual program can be executed, as it will appear that the system has the host's hardware such as processor, memory and other resources. In reality, however, it is the hypervisor that allocates these hardware resources to the VMs. An efficient hypervisor will let its users have several VMs, all of them working optimally on a single piece of computer hardware without interacting each other. Currently, this software is largely adapted to Linux and Unix systems. The use of virtualisation technology expands hardware capabilities, extend the extensive utilisation of components, control costs and improve reliability and security of systems especially where resources are limited in a platform such as cloud servers. A hypervisor that accesses all physical devices residing on a server, can also access its memory and disks. It can control all aspects and parts of a VM. Nevertheless, the hypervisor is what controls and allocates what portion of hardware resources each OS should get as they needed, without disrupting each other. The present-day hypervisors are the fundamental components of any virtualisation system. It can be referred to be the OS of a virtualised system [4]. CPU vendors now adding hardware virtualisation to their x86-based products, extending the availability (and benefits) of virtualisation to PC- and server-based audiences.

2.1.1 Types of hypervisors

A VM can create requests to the hypervisor through a variety of methods, including API calls. There are two types of hypervisors:

Type-I or Bare metal hypervisors

These type of hypervisors can run directly on the host's hardware to control them and to manage guest OSs. They are also called bare metal or native hypervisors. The first hypervisors, which IBM developed in the 1960s, were native hypervisors. Among other type-I hypervisors, VMware ESX and ESXi offers most advanced features and scalability but require licensing [5]. Lower cost of VMware can make hypervisor technology more affordable for small infrastructures. It is the leading type-I at present. Microsoft Hyper-V does not offer many of the advanced features but with the help of XenServer and vSpere, it is one of the best type-I hypervisors at present. Another example of a type-I category is Citrix XenServer and Oracle-VM which has open-source core hypervisor but lacks most advanced features like ESXi.

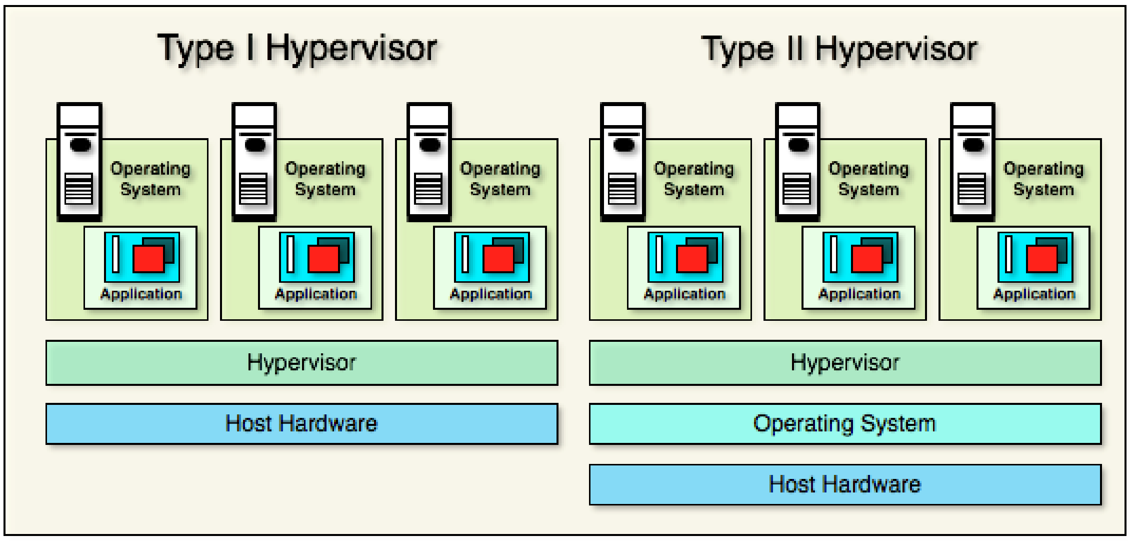


Figure 2.1: Illustration of the Difference Between Type-I and Type-II Hypervisors

Type-II or Embedded hypervisors

These hypervisors can run on a conventional OS just as other computer programs do. A guest OS runs as a process on the host. Type-II hypervisors abstract guest OSs from the host OS. They are also referred as embedded or hosted hypervisors. VMware Workstation, VMware Player, Virtual Box, Parallels Desktop for Mac and QEMU are examples of type-II hypervisors. VMware workstation has major use cases for running multiple different operating versions or systems on a single desktop. For the developers, this needs sandbox environment and snapshots and only can be used for labs and demonstration purposes. VMware servers, on the other hand, is free to use and a hosted virtualisation hypervisor. Microsoft Virtual PC is the latest version of hypervisor technology and has only limited options (e.g., can only run Windows 7). Oracle VMs should also be mentioned as a type-II hypervisor with reasonable performance and features of virtualised systems.

Finally, there is the Red Hat Enterprise virtualisation is the kernel-based VM (KVM) which qualifies both to be a bare-metal and embedded hypervisor. It has the ability to turn the Linux kernel itself into a hypervisor so the VMs can have direct access to the physical hardware itself.

2.1.2 Benefits of hypervisors

There are countless advantages of using hypervisor in a system. Some are mentioned below:

- VMs can easily run on the same physical hardware but they are usually separated logically from each other. This means that even though one VM experiences malware attack, error or crash at some point, it will not affect VMs on the same machine.
- VMs are mobile and independent of the underlying hardware. This means they can be migrated between local and remote virtualised servers in a lot easier way than traditional applications that are running on the physical hardware.
- Hypervisors can also be used in data services for easy cloning and replication.
- Hypervisor-based replication is also more cost-effective and less complex than current replication methods, especially those involving VMs. Hypervisor-replication can save up on storage space. System-based replication method requires replication of entire volume of the VM which can take a considerable amount of storage space with several VMs.
- Hypervisor-based replication is also hardware neutral, i.e., any duplicate data can be stored in any storage.

Considering the consolidating of the virtualised system, VMware, Microsoft and Citrix Systems are the three of the biggest vendors of the hypervisor in the enterprise data centre space.

2.2 Virtual Machines and Virtual Infrastructure

As discussed earlier, VMs are computer software that is like a physical computer, run an OS and applications. The end user has the same experience on a VM as they would have on dedicated hardware. The VMs are comprised of a set of specification and configuration and is backed by the physical resources of a host system. It usually has virtual devices which provide the same functionality as physical hardware and have an additional benefit in terms of portability, security and manageability. Specialised software like the hypervisor, emulates the PC client or server's CPU, memory, hard disk, network and other hardware resources completely, enabling VMs to share those resources. The hypervisor can emulate multiple virtual hardware platforms that are isolated from each other, allowing VMs to run Linux and Windows Server OSs on the same underlying physical host. It can have several types of files and even a storage to store those files. The configuration files, NVRAM setting file and the log files are the key files to make a virtualised machine. It limits the costs since it reduces the quantities of hardware maintenance costs and also reduces power and cooling demand. Administrators can take advantage of virtual environments to simplify backups, disaster recovery, new deployments and basic system administration tasks. Virtualised systems do not require specialised, hypervisor-specific hardware. Virtualisation does, however, require more bandwidth, storage and processing capacity than a traditional server or desktop if the physical hardware is going to host multiple running VMs.

The infrastructure that supports VMs consists of at least two software layers. They are the virtualisation and their management. In vSphere, ESXi provides the virtualisation capabilities that aggregate and present the host hardware to VMs as a normalised set of resources. VMs can run on an isolated ESXi host or on ESXi hosts that vCenter Server manages. vCenter server lets the user create resource pools and manage multiple hosts by effectively monitoring the physical and virtual infrastructures. In the vCenter Server hierarchy, a data centre is the primary container of ESXi hosts, folders, clusters, resource pools, vSphere vApps, virtual machines, and so on. Data-stores are virtual representations of underlying physical storage resources in the data centre. A data-store is the storage location (for example, a physical disk or LUN on a RAID, or a SAN) for VM files. Data-stores hide the idiosyncrasies of the underlying physical storage and present a uniform model for the storage resources required by VMs. Fig. 2.2, shows the virtualisation map.

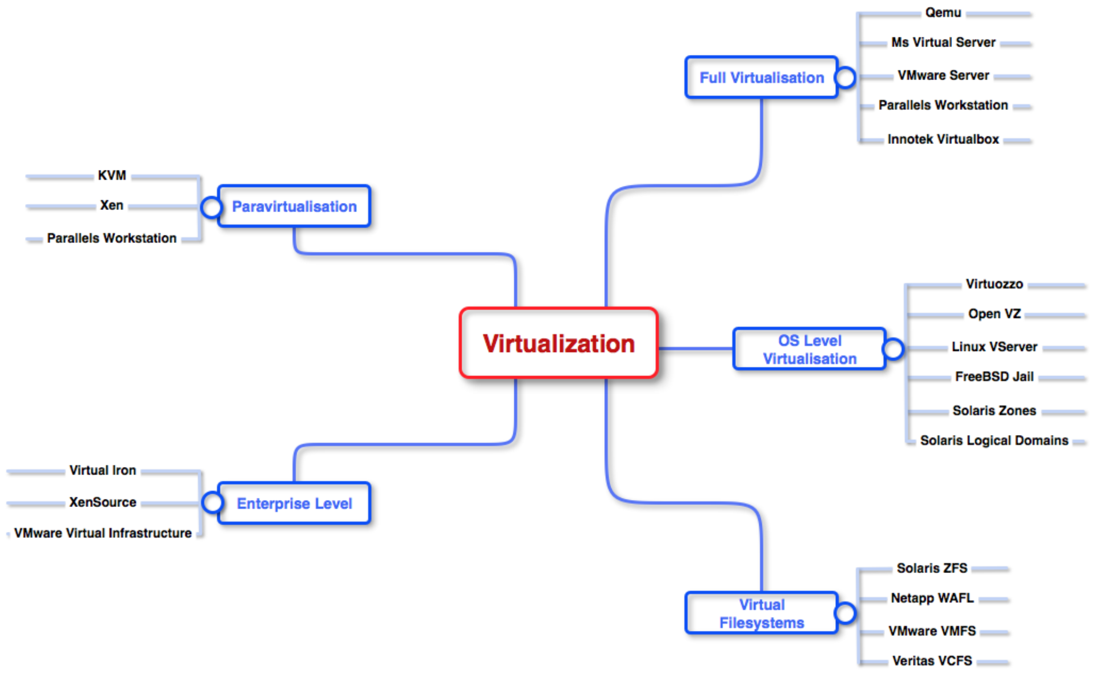


Figure 2.2: Illustration of Virtualisation Map

2.2.1 Kernel-Based Virtual Machine (KVM)

KVM is a virtualisation infrastructure for the Linux kernel that supports native virtualisation on processors with hardware virtualisation extensions. A wide variety of guest OSs works with KVM, including many flavors and versions of Linux, BSD, Solaris, Windows, Haiku, ReactOS, Plan 9, AROS Research OS and OS X. In addition, Android 2.2, GNU/Hurd (Debian K16), Minix 3.1.2a, Solaris 10 U3 and Darwin 8.0.1, together with other OSs and some newer versions of these listed, are known to work with certain limitations. KVM originally supported x86 processors and has been ported to S/390, PowerPC, and IA-64. An ARM port was merged during the 3.9 kernel merge window.

KVM does not perform any emulation by itself, instead, it exposes the /dev/kvm interface, which the set up the guest VM's address space can be used by a userspace host. The host must also supply a firmware image, usually a custom BIOS when emulating PCs, that the guest can use to bootstrap into its main OS. It can also feed the guest simulated I/O and map the guest's video display back onto the system host. On Linux, QEMU versions 0.10.1 and later is one such userspace host. QEMU uses KVM when available to virtualise guests at near-native speeds but otherwise falls back to software-only emulation.

2.2.2 Quick Emulator(QEMU)

QEMU is the software that actually creates the hardware which a guest operating system runs on top of. Depending on the QEMU configuration the virtual hardware is created which shows the device like a keyboard, mouse, network card, etc., to the guest OS. These devices are based on the actual specifications that are released for physical hardware available. QEMU mimics the real hardware and creates some devices written specifically for virtualisation use-cases. QEMU VMs usually use virtio framework for creating these devices. Virtio-net networking, virtio-blk block, virtio-scsi SCSI and similar devices created with QEMU. Para-virtualised devices have the benefit of being designed with virtualisation in mind, which makes them faster and easier to manage than the emulation of real devices. It uses several services from the host Linux kernel, like using KVM APIs for guest control using the host's networking and storage facilities. QEMU can also interact with other projects such as SeaBIOS that provides BIOS services to the guests.

2.2.3 Xen

Xen is a VM monitor tool for the x86 architecture. It is an open source software released under the GNU General Public License (GPL) and developed at the University of Cambridge. This software is a para-virtualisation technology which provides a bit modified hardware interface to the VMs. It works by separating hosts (DomainO) and guests (DomUs) OS into different parts called domain, that can run on top of a special hypervisor hardware interface. DomainO (privileged) usually handles the system commands (e.g., create, shutdown, reboot, etc) for the DomUs (unprivileged). One of the best features of the Xen is the live migration which allows continuous service operation by moving VMs to other physical hardware. It also provides load balancing of VMs, to a high-performance server from a highly congested server. Xen live migration reduces some of the complexity in the configuration since entire OS and all the applications are migrated as one unit. Live migration has two main requirements: shared storage and similar CPU architecture.

Users interact with virtual machines via one of the several available interfaces, like virt-manager, oVirt, OpenStack, or Boxes. These software interacts with libvirt (provides a hypervisor-neutral API to manage VMs). This also has APIs to interface with different blocks/storage and network configurations. In QEMU or KVM, VMs libvirt communicate with QEMU using APIS provided by it. A QEMU instance is present on each VM created on a host. The guest runs as a part of the QEMU process. Virtual CPU (vCPU) in each guest is then seen as a separate thread in the host's processes. QEMU usually interfaces with Linux, especially with the KVM module within Linux. This makes the VMs run directly on the physical hardware (not emulated by QEMU). Fig. 2.3, shows thread (one at a time) running QEMU code at any time on the hardware emulator and generation of I/O requests on IOThread to the hosts on guest's behalf and handles events.

2.3 Ubuntu

Ubuntu is a Linux distribution based on the Debian architecture. It is usually run on personal computers and is also popular on network servers. Most popular architectures such as

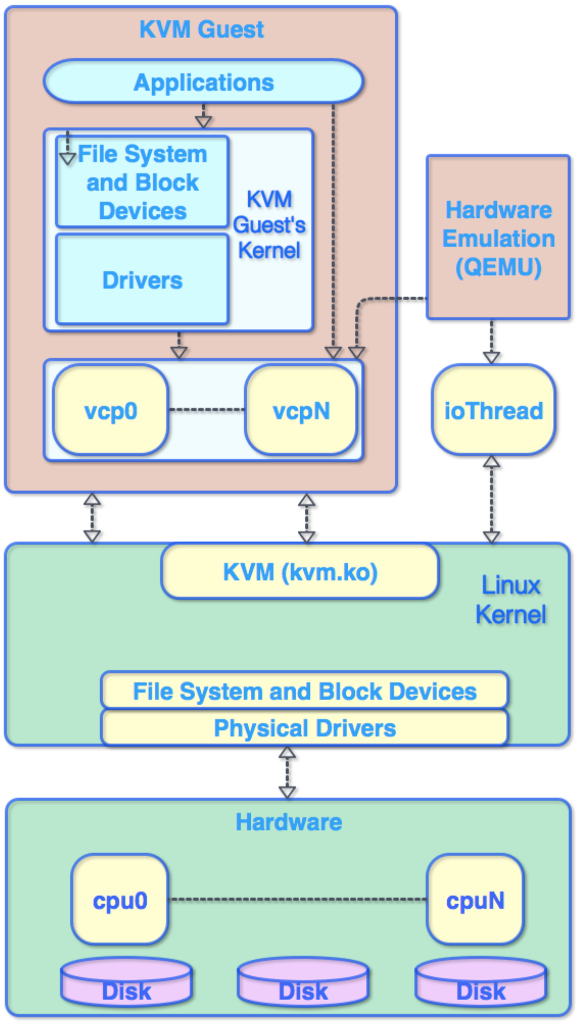


Figure 2.3: Illustration of Virtualisation Stack

Intel, AMD, and ARM-based machines usually run Ubuntu. This is the most popular operating system running in hosted environments, so-called "clouds", as it is the most popular server Linux distribution. It is published and developed by Canonical Ltd and based on free software. Ubuntu has a server edition that uses the same APT repositories as the Ubuntu Desktop Edition. The differences between them are the absence of an X Window environment in a default installation of the server edition and some alterations to the installation process. The latest server edition supports hardware virtualisation and can be run in a VM, either inside a host operating system or in a hypervisor as we discussed in Sec. 2.1. The AppArmor security module for the Linux kernel is as default on key software packages and the firewall is extended to common services used by the OSs The current Ubuntu release supports Intel x86 (IBM-compatible PC), AMD64 (x86-64), ARMv7, ARMv8 (ARM64), IBM POWER8, IBM zSeries (zEC12/zEC13), and PowerPC architectures. In this project, one of the most stable latest version of Ubuntu (Ubuntu 16.04.4 LTS (Xenial Xerus)) has been used for instance image platform.

2.4 Cloud Computing

Cloud computing is now adapted to all shapes and sizes of companies and industries. It is undoubtedly beneficial for users to use cloud platforms for as infrastructure (IaaS), platform (PaaS) or services (SaaS). Cloud computing is an information technology (IT) paradigm that enables ubiquitous access to shared pools of configurable system resources and higher-level services that can be rapidly provisioned with minimal management effort, often over the Internet. Cloud computing relies on sharing of resources to achieve coherence and economy of scale, similar to a utility. Third-party clouds enable organisations to focus on their core businesses instead of expending resources on computer infrastructure and maintenance. Advocates note that cloud computing allows companies to avoid or minimise up-front IT infrastructure costs. This type of computing allows enterprises to focus to get their applications up and running faster, with improved manageability and less maintenance. This enables IT teams to be more rapidly adjust resources to meet unpredictable and fluctuating business demand. If used properly and to the extent necessary, working with data in the cloud can vastly benefit all types of businesses. Mentioned below are some of the advantages of this technology [6]:

2.4.1 Benefits of cloud computing

- Cost efficiency is the best feature of adapting to cloud computing which benefits all type businesses. Instead of using traditional hardware and software to run a infrastructure for a business, it is considered to be a fraction of a cost while running it in the cloud. IT expenses are significantly low while cloud computing is adapted to the businesses.
- Storing information in the cloud gives almost unlimited storage capacity. Running out of space is no more a concern.
- Since all the data is stored in the cloud, backing and restoring is relatively much easier than storing the data in a physical device. As a fail-safe most cloud providers are equipped with a recovery handling facilities.
- Software integration is usually that can be done in the cloud automatically. This takes away additional effort to customise and integrate deployed applications in the systems.
- Services and software application can be picked that is best suitable for business or personal use.
- Most importantly, the ease of accessibility the VMs in the cloud from geographically anywhere make it an unbeatable technology of the generation.
- Cloud computing gives the advantage of quick deployment, i.e., the system can be deployed and be fully functional in few minutes depending on what technology is used.

2.5 Overview of the Tools

Based on features enabled by virtualisation system, cloud service components and inspiration behind the idea of the autonomy of services, we researched and utilised the following tools in our methodology and design approach. Some of the main features of the tools, which helps to choose for this project, are as follows:

2.5.1 Google Cloud Platform (GCP)

Google Cloud Platform is a set of physical components (computers and hard disk drives) and virtual resources (VMs), that are contained in the Google's data centres around the globe. The location of each data centre location is in a global region. This region includes Western Europe, Central US and East of Asia. Each region has a collection of zones, which are isolated from each other and identified by a unique identifier (e.g. us-central1-f). The benefit of this kind of distribution is such as including redundancy in case of failure and reduced latency by locating resources closer to clients. The distribution describes how the resources are used together.

GCP HTTP(s) load-balance

GCP HTTP(s) load-balancing provides global load-balancing for HTTP(s) requests destined for the instances in GCP. URL rules can be configured to route some URLs to one set of instances and route other URLs to other instances. Requests are always routed to the instance groups that is closest to the user, provided that group has enough capacity and is ready to handle the requests. If the closest groups capacity exceeds the limit, the request is then sent to the closest group that does have the capacity. This type of load balancing supports both IPv4 and IPv6 addresses for client traffic. HTTP requests can be load-balanced based on port 80 or 8080 and for HTTPs it can be port 443. The load-balancer acts as an HTTP/2 or HTTP/1.1 translation layer, which means that the web servers always see and respond to HTTP/1.1 requests [7]. In cross-region load-balancing, a global IP address can be used to intelligently route users requests based on proximity. This means the back-end servers located geographically closest to the users will be getting requests first automatically until the servers' capacity runs out. If the closest instances do no have enough capacity, cross-region load-balancing forward users request t the next closest region.

Google cloud DNS

Google Cloud DNS is one of products of Google Cloud Platform (it's placed under Network products) that allows us to manage Domain Name System (DNS) records for any domain. DNS stands for Domain Name System. DNS is a hierarchical decentralized naming system for computers, services, or a resource connected to the Internet or a private network. In more simpler words, when a URL of some website is typed in the browser, the computer knows to which IP address of the server it needs to connect in order to obtain data for the website. A good analogy will be "Phonebook", but instead of a name of the person, we need to have web domain or webpage. And instead of phone numbers, it needs to have the IP address of the servers. So every time the URL is called, DNS server looks IP address of the server which is connected to the domain of that webpage. There are different DNS records with which domain can be managed. Each record has its purpose, some type of DNS records are:

- A - IPv4 address, maps IP of the host with a domain.
- CNAME - alias for one name to another, for example, www.example.com points to example.com.
- MX - manages where emails should be delivered.
- TXT - whatever text, for example, Google uses it to verify that the ownership of the domain.

There are much more important thing to emphasize is that, to lower the load of DNS servers, DNS records are cached for certain time (which can be set with TTL parameter), so when some changes are done, it can take some time to reflect. Although we can manage DNS records within any domain registrar account, there are some advantages of using Google cloud DNS:

- It is cheap to use and easy to maintain.

- It is on the same network as our Google compute engine, so requests have smaller latency.
- It is fast, scalable, reliable and secure as it runs on its infrastructure which is managed by Google.
- Possibilities to manage DNS records through Gcloud SDK, REST API, cloud console etc.
- Cloud DNS uses Google's own global network of Any-cast name servers to serve DNS zones from redundant locations around the world, providing high availability to any users. Anycast addressing is a one-to-one-of-many association where datagrams are routed to any single member of a group of potential receivers that are all identified by the same destination address. The routing algorithm selects the single receiver from the group based on which is the nearest according to some distance measure. Anycast is not specifically supported by IPv4, but this omission can be worked around in many cases by using Border Gateway Protocol (BGP). Multiple hosts (usually in different geographic areas) are given the same unicast IP address and different routes to the address are announced through BGP.
- A complex project as this masters thesis can have multiple domains, sub-domains. In this case, Google cloud DNS is useful.

2.5.2 Python

As the programming tool, the Python programming language is used as our software platform to design our protocol. It is a widely used general-purpose, interpreted, high-level, dynamic programming language. Its design philosophy gives priority to code reliability. The syntax of Python allows programmers to express the concept in fewer lines of code than the languages such as Java or C++. This language helps to get a neat and clear program in both large and small scales. It supports multiple programming paradigms which include object-oriented, functional procedural styles. This language can be packaged as standalone executable program most operating systems. This allows this programming language to be running on any operating environments without any help from Python interpreters. Python is a free and open-source software, run by a non-profit software foundation, which features a dynamic type system and automatic memory management and has come with late and comprehensive standard library. In our project, we use Python v3.5.4 to build our design and code implementations [8].

2.5.3 Bash

Bash is a Unix shell and command language written by Brian Fox for the GNU project as a free software replacement for the Bourne shell. It is first released in 1989 and has been distributed widely as the default login shell for most Linux distros and Apple's macOS. The tool is a free software and complies to UNIX standards. Bash is a command processor that typically runs in a text window, where the user types command that cause actions. The language can read and execute commands from a file called a shell script. Like Unix shells, it supports filename globbing (wildcard matching), piping, command substitution, variables and control structures for condition testing and iteration. Bash is a POSIX compliant shell, but with a number of extensions. The syntax is a superset of the Bourne shell command syntax. It can execute the vast majority of Bourne shell scripts without modification and with the exception of Bourne shell scripts stumbling into fringe syntax behaviour interpreted differently in Bash or attempting to run a system command matching a newer Bash built-in, etc. Bash 3.0 supports in-process regular expression matching using a syntax reminiscent of Perl. Some of the best features of Bash are the ability to run startup scripts, conditional execution, shell arithmetic, aliases, directory stack, arrays, process management and controlling the prompt.

2.5.4 HAProxy

HAProxy is single-threaded, event-driven, non-blocking engine combining a very fast I/O later with a priority-based scheduler. The architecture is designed to move data as quickly as possible with the least possible operation keeping in the forwarding of data in mind. Data does not reach the higher level in OS model while performing offering a bypass mechanism at each level. HAProxy let the kernel do that most processing work and avoid certain operation when it guesses they could be grouped later. Typically 15% of the processing time spent in HAProxy versus 85% in the kernel TCP or HTTP close mode. Also, the HTTP keep-alive mode is about 30% for HAProxy versus 70% for the kernel [6]. A single process in HAProxy can run as much as 300000 distinct proxies which require only one process for all the instances. It is also possible to run it on multiple processes. Usually, this tool scale very well for HTTP keep-alive mode but the performance that can be achieved out of single process generally outperforms common needs by an order of magnitude. HAProxy only requires the haproxy executables and a configuration file to run which makes it easy to use for the service providers. a syslog daemon needs to accurately be configured for logging services and rotation services for logs. The configuration files are parsed before stating and HAProxy tries to bind all the listening sockets. If anything fails then it refuses to start. The run-times failure is next to none once HAProxy accepts to start.

HAProxy is an open source software used for TCP/HTTP load balancing. It runs on Linux, FreeBSD, and Solaris. It is famous and widely used for its ability to keeps servers up by distributing the load across multiple servers. Among many terminologies used in HAProxy Access Control List (ACL) contains a set of rules which has to be checked so that it can carry out some predefined actions which can be blocking some requests or selecting the server to forward the request based on the conditions. ACL rules apply to all the incoming traffic which increase the flexibility of the traffic forwarding based on different factors such as some connection to the back end and pattern matching. A back-end is the group of servers that have been established for load-balancers. It contains the IP addresses of the servers along with the port numbers)(if necessary) and chooses the load-balancing algorithm for efficient processing of the web-server access requests. Another attribute of HAProxy is front end where the configuration defines how the requests have to be forwarded to the back-end servers with ACL. The definition of front end contains IP addresses and port numbers of the servers as well. HAProxy also has a health check-up feature which in a simple way checks the back-end servers' health. It is carried out by simply sending a TCP request and find whether server listens to the specific ports and IP. Upon no response, the load-balancer can fire up a forwarding request to another server which is healthy. The unhealthy back-end server does not get any further request until it checks up as a healthy server again. At Least, one server should be healthy to process the request.

Among many load-balancing algorithms in HAProxy, the commonly used algorithm is round-robin algorithm. The algorithm chooses the server sequentially in the list. Once it reaches the end of the server list, the algorithm forwards the next request to the first server in the list again. The weighted round-robin algorithm uses the weight allocation to the server to forward the request while dynamic round-robin algorithm uses the real-time updated weight list of the server. The least connection algorithm is also being used in HAProxy which selects the server with few active transactions and then forwards the user requests to the back end. The source algorithm also selects the server based on source IP addresses using the hash to connect to the machinating server. Server overloading can be reduced using high availability algorithm which gets activated when the primary (active) load-balancer gets overloaded. It fires up the secondary (passive) load-balancer if the primary load-balancer fails. In this project, a primary load-balancer is used with round-robin algorithm in order to forward all the incoming request to the back-end server [9].

2.5.5 Consul

Consul is a software designed by HashiCorp is a distributed, highly available system. It is a complex system that has many different moving parts. This tool has multiple components for service discovery and configuration in the infrastructure. The key features of this tool are service discovery, health checking, key/value storing for any number of purposes and supporting multiple data centres out of the box. The architecture of Consul consists of following terms:

- **Agent**: An agent is the long-running daemon on every member of the Consul cluster and started by running consul agent. The agent is able to run in either client or server mode. All nodes must be running an agent and usually refer to the node as being either a client or server as well other instances of the agent. All agents can run the DNS or HTTP interfaces, and are responsible for running checks and keeping services in sync.
- **Client**: A client is an agent that forwards all remote procedure calls (RPCs) to a server. The client is relatively stateless and takes part in LAN gossip pool as the part of only background activity. This has a minimal resource overhead and consumes only a small amount of network bandwidth.
- **Server**: A server is an agent with an expanded set of responsibilities including participating in the Raft quorum, maintaining cluster state, responding to RPC queries, exchanging WAN gossip with other data centres, and forwarding queries to leaders or remote data centres.
- **Data centre**: While the definition of a data centres seems obvious, there are subtle details that must be considered. A data centres are considered to be a networking environment that is private, low latency, and have high bandwidth. Each data centre recommends running at least one Consul server.
- **Consensus**: Consensus is to meant as an agreement upon the elected leader as well as agreement on the ordering of transactions. Since these transactions are applied to a finite-state machine and implies the consistency of a replicated state machine.
- **Gossip**: Consul is built on top of Serf which provides a full gossip protocol that is used for multiple purposes. Serf provides membership, failure detection, and event broadcast and involves random node-to-node communication, primarily over UDP.

Every node that provides services to Consul runs a Consul agent and does not require for discovering other services of getting/setting keys or value data. The agent is responsible for health checking of the service on the node as well as the node itself. The agents can talk to the node or more Consul service where the data is stored and replicated. The servers themselves can elect a leader. Components of an infrastructure need to discover other services or nodes and can query any Consul servers or any of the Consul agents. The agents can forward queries to the servers automatically. When a cross data centre service discover or configuration is made the local Consul servers can forward the request to the remote data centre and return the result. Applications can make use of Consul's hierarchical key/value store for any number of purposes, including dynamic configuration, feature flagging, coordination, leader election, and more using simple HTTP APIs. This tool is designed to support both DevOps community and application developers which makes it perfect for modern, elastic infrastructure.

Along with other important tools mentioned above, following is used for benchmarking and testing purposes.

2.5.6 ApacheBench (AB)

Load testing is a good idea before any production deployment. It's nice to quickly establish a best-case scenario for a project before running more detailed tests down the road. The ApacheBench tool (ab) can load test servers by sending an arbitrary number of concurrent requests. This especially shows how many requests per second the Apache installation is capable of serving. This is a tool for benchmarking any Apache HTTP server. It is designed to give an impression of how the current Apache installation performs. AB is a single-threaded command line computer program comes bundled with the standard Apache source distribution, and like the Apache web server itself is free, open source software and distributed under the terms of the Apache License. ApacheBench will only use one operating system thread regardless of the concurrency level (specified by the -c parameter). In some cases, especially when benchmarking high-capacity servers, a single instance of ApacheBench can itself be a bottleneck. When using ApacheBench on hardware with multiple processor cores, additional instances of ApacheBench may be used in parallel to fully saturate the target URL.

2.6 Related Work

Many interesting research work have been done on VM migration, self-awareness and automation of a VM in cloud infrastructure. Many approaches have been adapted to be utilised in the field of research and some are already in use in operation in most well-known service and cloud providers. Other approaches are slowly coming out of the surface of under-development layer to the product ready level. In this project, we are mainly focusing on the algorithms and their approaches in practical and theoretical work. Finding the right algorithm in order to make our VMs more autonomous and management-free approach in the runtime compilation of the services is one of our fundamental concept in the system design. It is not easy to find one specific concept that is suitable for the VM's optimal performance as completely independent and minimising network complexity at the same time optimising faster and achieve emergence behaviour of the deployed VMs are tended to be a difficult and demanding process. Most research approaches are generally focused on the centralised approaches of the VM management where a single system are in charge of other smaller VMs which providing services by observing the network behaviour and sending/receiving information in order to manage those VMs. This is a cluster concept where controller VMs can become the bottleneck of the infrastructure. This means these systems are vulnerable to become depleted or disconnected from the network which can make the whole cluster become incapable of responding to network demands. Some researches are done where it is targeting the cluster of VMs to be scalable as the need for the services increases. This requires the unnecessary use of components when many VMs can be created just to respond to the incoming request in a single cluster especially in the cloud platform when the VMs are separated in different zones or locations. The following are the brief description of various categories of related work that we find most interesting in our early research work.

2.6.1 Researches based on evolutionary game theory

This masters thesis is inspired by various algorithms those are mainly based on natural phenomenon. Most of the legacy algorithms, implemented in the cloud computing are motivated by observing the nature and its properties, especially the evolution of nature and animal behaviour as a collective system. Some of the interesting topics are discussed in the following section.

The researcher of paper [10] has brought our attention to the evolution algorithm for implementing a behavioural feature on the VMs to pick the most popular video clip for the mobile video users. The author of this paper pointed out that multiple video users could share the same video clips based on content reused. Video users are watching those video clips that most popular and have the best quality. They advocate that the video applications are provided by many well-known video provider (such as YouTube) serve with contents that have distinctive characteristics. Based on content reuse the quality of the video differs from each other by the number of the user request to access those videos. More popular the video is better the quality it serves. High definition videos tend to get more view due to their popular demand. The problem regarding video's services is addressed in fewer work than the research on energy-spectrum-aware scheduling scheme for video streaming. The researchers contribute with an evolutionary algorithm which based on the content reuse of the video by observing mobile video user (MVUs) behaviour and popularity of the video as the utility of every MVU. The replicator dynamics in the evolutionary game is utilised to model the interaction of MVUs. According to there algorithm, MVUs are given a number of choices of video clips. Like an evolution game, all the players are grouped together into a popularity. Each player would independently choose a certain strategy, especially the strategy providing the highest quality of video. An evolutionary equilibrium is in place where there is no change in the proportion of players choosing different strategies. The algorithm runs as simply by each MVU to get access to a video clip in random where the utility of MVUs accessing video clips is calculated. Then it is checked if the utility of each MVU is is less than the average utility of all the video clips. If true then the MVUs can choose another random utility until it finds the one has the better than average utility. These steps continue until all the best quality video clips are found by MVUs. The authors also proposed a Q-learning based evolution algorithm which helps to avoid exchanging too much information and facilitates each MVUs with the ability to work without the knowing the strategy of other MVUs. The paper concluded by dealing with the video clip selection process by considering relevant influential factors. In this project, we can use the similar concept of the evolutionary game algorithm on our servers with autonomous VMs to implement self-organising or self-managing behaviour.

Another theoretical research work [11] we review where the authors discuss the issue regarding the deployment of degradation of performance of the big data applications in the cloud servers. They mentioned that using a traditional fixed-resource allocation mechanism has a couple of drawbacks such as low resource utility and unresponsiveness to the performance degradation. The resource here was mainly referred to the combination of CPU, memory, I/O and network resources. To address those drawbacks the researchers proposed a hybrid environment for cloud and big data where resource allocation is used fairly to ensure fairness between cloud and big data application. They also proposed the approach to migrate VM in order to make each VM in cloud application to reach a certain level of efficiency. They start by allocating resources dynamically to the cloud or big data applications and used the game theory to model this resource allocation problem to ensure fairness. The cloud and applications were assumed to be stable compared to big data applications as no new cloud tasks will come and finish while big data processing tasks under a certain period of time. The application in the cloud is set to provide services to the users (e.g., web servers, FTP servers, etc.) and big data applications are set to run computing tasks and run for more hours. Thus, adapting to dynamic resource allocation was necessary to avoid conflict and negotiation between these types of applications. The VM migration is set to maximise the utility of each VM while the minimal utility is guaranteed. The project adapted Nash bargaining game to model the VM migration problem. How the theory works is by finding the best strategy combination for all the other players (VM) under equilibrium situation which leads to the maximum utility achievement for each VM. In this case, each VM's best strategy will rely on the other VM's strategy. This solution with Nash equilibrium works very well to maximise fairness in resource allocation for the applications. The migration mechanism for VM was designed to try making them meet their own demand within the limited resources. Using the above approach the authors of this paper demonstrate that the resources utility and average performance of the cloud and big data applications were much higher in their designed hybrid environment than the traditional environment. The resources could be rearranged to satisfy the applications that lack resources.

In the paper [12], the researchers call reader's attention to the issue regarding the network-driven approach of a load-balancing heterogeneous network. The adaptation of next-generation wireless networks will integrate the wireless access technology to provide seamless mobility to mobile users with high.speed wireless connectivity. Network selection of load-balancing will also become crucial as it will try to avoid network congestion and performance degradation. As the different service areas have limited amount of bandwidth available to share among a group of wireless users the evolution game theory is considered to be a more fitting solution to this setup. Load-balancing in a heterogeneous network can be achieved by using either a user-driven or network-driven approach. For the user-driven load-balancing approach, network-selection algorithms are implemented at user mobile that may be preferred due to its low implementation complexity and low communication overhead. In the latter approach, a centralised controller assigns network resources to the connections in a service area which on demand to keep all the available wireless networks tightly integrated. This can result in a large communication overhead. The authors used a dynamic evolutionary game with multiple populations to analyse the behaviour of the users in the network selection. The game was formulated to logically model the competition among groups of users in the different service areas with various types of wireless technologies such as WMAN, cellular network, etc. The evolutionary equilibrium was targeted as their solution for competition. There was two algorithm presented in this paper to obtain a solution, population algorithm (uses information about the users in the corresponding service area) and the reinforcement learning algorithm (only utilises local knowledge obtained through learning to reach the evolutionary equilibrium). Finally, the solution to the network-selection problem obtained from the evolutionary game model is compared to the Nash equilibrium solution obtained from a classical non-cooperative game model. A centralised and a distributed algorithm were proposed to implement the proposed evolutionary game model for network selection. In their experiments, they found out that their proposed network-selection based population evolution algorithm takes much less convergence time than the reinforcement learning algorithm in which a user selects a network independently by using its local payoff information obtained through exploration. The developers have investigated the dynamics of network selection in heterogeneous wireless networks using the theory of evolutionary games and came to a conclusion that their evolutionary equilibrium model had been considered to be the stable solution for which all users receive an identical net utility from accessing different networks.

This particular paper [13], as the previously mentioned related work, also focused on developing a resource allocation algorithm based on the evolutionary game theory on Mobile Cloud Computing (MCC). The research considered mobile terminals' energy consumption and time delay as well as monetary cost in mobile edge computing environment in the wireless network. As the researchers specified that MCC is a significant paradigm that combines wireless network service and cloud computing to enable mobile terminals to enjoy the abundant wireless resource computational power ubiquitously but partially suffers from limited battery power and scarce computing capabilities. It can also introduce high latency because of the distance between terminals and powerful servers are geographically far. Mobile edge computing is vastly adapted to solve this issue but this does not solve the limited radio and back-haul communication capabilities. In this paper [13], the scientists put forward a joint cloud and wireless allocation algorithm, where the mobile terminals which have the tasks offloading requirements in different service area form a population. Cost function in the game model intends to measure the consumption of tasks offloading and is involved in the energy consumption, monetary cost and time delay. The model formulation of their game algorithm is modelled by taking advantages of a dynamic evolutionary game. The formulation of the game consists of players (mobile terminal chooses macro and small service providers), population (set of terminals in the same service area), strategy (selection of each service provider). population share (number of mobile terminals selecting service providers for population), population state (population shares of all service providers) and finally the cost function (to measure the affected cost of a player i.e., energy consumption, time delay and monetary cost). Using their replicator dynamics and equilibrium and stability analysis of evolutionary game algorithm this research provided simulation results that verified that the effectiveness of their algorithm can save more energy and have less time delay when the input data size becomes larger, compared to existing algorithms.

2.6.2 Researches based on the migration of VMs

Migrating operating system instances across distinct physical hosts is a useful tool for administrators of data centres and clusters: It allows a clean separation between hardware and software; and facilitates fault management, load balancing, and low-level system maintenance.

In this paper [14], the scientists came up with an exquisite solution to migrate the VMs to a different server while the OSs continues to run. This means the live migration and was done by migrating the entire OS and all of its applications as one unit. This allows avoiding many of the difficulties that arise by the process-level migration process. The interface between a virtualised OS and VM monitor (VMM) makes it easy to avoid the problem of residual dependencies in which the original hosts always available and accessible by the network in order to service certain system calls or memory access for the migrated process. The authors describe that VM migration can decommission the original host once the migration is completed. Additionally, migrating the level of an entire VM means that in.memory state can be transferred in a consistent and efficient way that applies to the kernel-internal and application-level state. The live migration allows a separation of concerns between user s and operator of a data centre or clusters. It allows the separation of hardware and software considerations and consolidating clustered hardware into a single coherent management domain.

This interesting research work yields a technique to implement high-performance migration support for Xen and their design and implementation addresses the issue and trade-offs involved in live migration. One of the issues was that minimising the downtime of active OSs hosting live services and other was mentioned to be the total migration time during which state on both source and destination machines is synchronised and might affect the reliability. The entire project was ensuring the migration itself does not disrupt the active services through the resource contention with migrating OS in an unnecessary manner. Their memory transfer technique push phase (source VM push the pages to destination across network while continues to run. Modified pages were pushed again), stop-and-copy phase (source VM stops, pages gets copied to destination, new VM starts) and pull phase (new VM executes and pull the accesses the pages that had not been copied). There were also pure stop-and-copy, pure demand-migration and pre-copy migration procedure were proposed. Using their self-migrating technique they make the VM to destination machines, by running a migration stub on the destination machine to listen for incoming migration requests, by creating an appropriate empty VM followed by receiving the migrated system state. By integrating live OS migration into the Xen virtual machine monitor they enable rapid movement of interactive workloads within clusters and data centres. Their implementations performed very well with minimal service downtime, about 60ms.

Live migration of VMs can have a major impact on cloud system performance and can consume a critical amount network bandwidth and other resources. The following research work [15] were focused on the autonomous network aware VM migration strategy in the cloud data centre. In this paper, the authors mentioned the issue with live migration which requires a considerable amount of cloud network resources. The goal of live migration is to balance the network resource with network traffic while satisfying the VMs and host resource constraints in the data centres. To ensure the QoS guarantees for both the customers and cloud providers, VMs must be relocated to another host that has sufficient resources and currently underutilised. Maintaining the migration from a host when the network traffic demand is in high level can be a challenging and complex strategy. Lack of free available resource can result in a long period of time for multiple migrations to be completed. This can take a toll on the performance degradation of VMs and affect the service level agreement of the host due to over or under utilisation of the host machines. The researchers claimed that a few research had been done in the area of time sensitivity regarding the migration of the VMs. Time is an important factor to consider when migrating VMs. Considering migrating VMs at varying network traffic demands could potentially increase or decrease the total time to move a VM from the source host to destination host. To handle these issue authors of this paper [15] proposed a migration strategy that observes the current demand level of a network and performs necessary decision based on it. The artificial intelligence technique called Reinforcement Learning act was used as decision support system which enables an agent or VM to learn network traffic demand and migrate accordingly if necessary. Their novel autonomous learning agent with the capabilities of deciding a suitable time to schedule a group of VMs for migration from an under-utilised host by analysing the current state of network bandwidth. The optimal action of the migrating agents was performed to utilise the resources that are available during live migration. The researchers introduced a resource utilisation pattern to the agents which run periodically for a week, day, hour and minute.

The autonomous agent will learn from this pattern and migrate the VMs by delaying the network resource consumption and ensuring the improvement of live migration. Their approach shows that it is possible to delay migration for the most opportunistic time for full utilisation of network resources.

Scalable cloud computing can be a suitable solution for most of the web service deployment in a cluster. But it is not always effective where the requirement of VM supporting those services becomes large, i.e., the number of VMs need to be increased in a large manner which can conclude in a significant amount in maintenance cost. Although the approach of this project is to maintain QoS of dynamically distribute web services by sharing the autonomous VMs by migrating to affected zones, some principle of using the metrics in cloud computers to achieve autonomy of the VMs can be interesting such as the methods adopted in this paper [16]. The researchers pointed out that computing infrastructure that is stable and provides reliable services is a key to the successful operation of complex contemporary enterprises. Especially when the information is demanded in real time and often on-the-go and the situation can get worse if the amount and usage information that are demanded are not predictable any more. The enterprise networks are costly and time consuming to prepare for such situations. The underlying technologies and issues for cloud computing, includes resource virtualisation, scalable resource management, load balancing of resources across time and location, and quality of service. The issues Cloud computing infrastructure entails are not entirely new but the long-standing fundamental problems of high-performance computing. Efficient assignment and scheduling of resources are yet to be developed to dynamically match the workload demands in the absence of the human intervention. Infrastructure administrators manually schedule the VM migration to sustain the fluctuating demands which result in a complex operation. VM migrations are an important underlying technology of IaaS for building efficient cloud computing infrastructure on a cluster of servers. It is designed to move VMs from an overloaded physical machine or zones to a lightly loaded machine lessen the burden on them while utilising the idling physical machines. The researchers of this paper [16] claim that migration decisions may not fit various different situations. In fact, some decisions may cause more unstable and unnecessary migrations. To solve this issue they introduce a learning framework that finds thresholds for machine overloading and underutilised dynamically. In their approach, if a particular computing pattern caused imbalance, that will trigger VM migration. This pattern will be saved or learned with its corresponding migration details for future use. The paper also proposed a proactive learning methodology that not only accumulates the past history of computing patterns and resulting migration decisions but more importantly searches all predefined possibilities for the most suitable decisions. The proposed autonomous learning system to learn environment overview to monitor and collect CPU usage statics by the kernel-level modules, normalising and identifying the CPU and memory load patterns by collecting the CPU status and memory information as raw data to find the unusual patterns with a standard deviation that indicates how far a particular CPU or memory usage is from the mean. By implementing *SBUML* into their dynamic migration framework module which provides a bridge between the host machine and the guest VMs to select a VM for migration when a host machine is overloaded, they have achieved their project goal. Their experiments demonstrate that their self-organised autonomous VM migration increases the resource utilisation of the servers on which the IaaS is running on. Especially, their proposed method resulted in an impact on learning new patterns of both CPU utilisation up to 20% from 5% while memory utilisation had gone up by 5-10%.

A good design to detect over-load and under-load of the host machine and placement algorithm for VM is one of the main focus of VM optimisation research. The research work done by scientists in the paper [17] have proposed a Markov prediction model to forecast the future load state of the host. Their work was motivated from observing the presence of the dynamic environment that leads to a changing load on the VMs. The authors explained that the fast growth of cloud computing it is crucial to effectively manage resource sharing and time access flexibility for applications and other components. Application models are required to assess the suitable amount of resources for each workload. Live VM migration migrates the entire OS and its associated applications from one host to another where a user does not notice any interruption in their service. Since in the modern day resource management and allocation during VM migration has become more challenging due to high dynamics of hosted services, high demands of services and resource elasticity, there is a need to reorganize the VMs and the hosts to provide load balancing or server consolidation depending on the Service Level Agreement (SLA) with the end users and other issues. To address three most important research problem in live migration, host over-load/under-load detection, VM selection and their placement, the researchers of this paper [17] proposed an algorithm which consider both current and future host utilisation states for VM placement, instead of only considering trade-offs between power consumption and SLA violation at the current host only. By combining the first-order Markov chain model was used to build Markov host prediction model along with a host load detection algorithm called Median Absolute Deviation Markov Chain Host Detection (MadMCHD) algorithm to find the future over-utilised/under-utilised host's state to determine suitable hosts in advance while live migrating VMs, the paper proposed an efficient algorithm to solve the issue. They evaluate the host detection model and implement their algorithm on a simulated large-scale data centre and compare the impact of data workload with the other well-known state-of-the-art host detection algorithms in the literature (namely *iqr*, *mad*, *lrr*, *lr*, and *thr*). The experimental results show that their proposed algorithms had a significant reduction in terms of SLA violation, the number of VM migrations and other metrics as compared to the most commonly used algorithms.

Similarly, the paper [18] presents a method to handle the highly variable user load from different geographical locations based on time-of-day, high or low growth rate, etc. The users requesting for different locations need the equal performance on QoS, so the researchers suggested Global Live Migration (GLM) of VMs using a software-defined network (SDN) enabled network infrastructure, because of its extensive control over network traffic routing. SDN controlled inter-site links can help move a VM from one data centre to another despite its geo-location. The researchers experimented on solving the problem of high chance in network congestion in inter-site links when several VM migration conducts in random order. The authors clarified that the VM migration can fail if a random migration sequence is followed without first assessing the capacity of each inter-site link using the bandwidth required for a successful migration of a VM. They maximise the number of successful live migration by choosing a near optimal migration sequence that helps the cloud service provider to maintain the response time for all the applications. In order to achieve that, they formulate a Mixed Integer Programming problem using Lagrangian Relaxation to find a near-optimal sequence of VM migrations that addresses the issue of network congestion and Self-Tuning Regulator, that leverage network traffic patterns as a feedback to enhance the optimisation model continuously. A feedback-based control system was implemented that leverage the network traffic data from the SDN controller to enhance the accuracy of our near-optimal solution in each iteration. The solution is evaluated by simulating it using the *CloudSimSDN* [19] simulator with a real-time scenario if cloud-based data centres hosting multiple applications. Their aim seek to avoid response time SLA violation for all of the hosted application which they achieved by successfully maximise the number of VM live migration and placing them closer to the user's geographical location, which proved to be a better solution in performance (minimum improvement of 38.09%) than other best fit heuristic algorithms.

Lastly, in this category of research work, authors of this paper [20], offer an extensive VM migration scheduler called *mVM*, to provide schedules with minimal completion times to schedule the migrations wisely when numerous VMs must be migrated. In order to minimise the impact on both the infrastructure and QoS, the mVM scheduler relies on realistic migration and network model to compute the best moment to start each VM migration and the amount of bandwidth allocated. mVM also decides which migrations are executed in parallel to provide fast migrations and short completion times. The scheduler was implemented as a set of extensions for the customisable VM manager *BtrPlace*. The mVM scheduler was performed against an unmodified version of *BtrPlace* maximises the migration parallelism and a scheduler that reproduces *Memory Buddies* decision by statically capping the parallelism. The migration model accuracy was also evaluated against representative cloud simulators models such as *CloudSim* and simplified the actual migration behaviour. The research's main results were concentrated on the prediction accuracy of mVM, migration speed of the scheduler, energy efficiency, scalability and extensibility of the proposed scheduling system through independent high-level constraints. The mVM was implemented as a plug-in of *BtrPlace* and its current library allows administrators to address temporal and energy concerns. Although their goal was to mainly compare the energy efficiency against state-of-the-art scheduler which it saves about 21.5% Joules against *BtrPlace*, mVM also reduces the individual migration duration by 20.4% on average and the scheduler completion time was reduced by 28.1% which is interesting for this masters thesis project research work as we are trying to accomplish an optimal VM migration design principle while maintaining integrity and QoS in overall cloud infrastructure.

2.6.3 Researches based on autonomous self-organising VMs

In the following paper [21], the writers address the problem with the resource sharing of an application by virtualisation of the systems arises when the resources such as CPU, memory, etc., lacks a management strategy. This management needs to be done autonomously, i.e., without any help of external instructions and by only observing the network and other agents that are connected. In a multi-core physical node, many VMs can run at the same time and share them by scheduling them. In many cases, these sharing schema does not help the VMs that need to the allocation of the resource more than normal amount of usage or less depending on how the VMs are being utilised. This demands a solution where resources needed by each VM adjusts according to the application demand. The author offered a solution to this case. They applied a policy that allows fulfilling the agreed application-level QoS and a mechanism, based on priority weight, to spread the unused resources between the contending VMs. And finally, a learning mechanism was proposed to dynamically reach and meet service level objectives (SLOs) expressed in transactions per second.

The priority weight approach allows to use the Xen's scheduler console configuration tool and change the proportion of the assigned resources. Using this weight a certain distribution of the resources (in this paper [21] CPU) for contending VMs are accomplished. The authors pointed it out that if a VM goes offline then the weighting mechanism assigns more resources to the active VMs which made it difficult to keep track of the appropriate weights that map an expected performance. With the controller approach, the management component includes the sensor algorithm and the controller algorithm. The algorithms run in parallel and communicate to each other by a queue. The algorithms are scheduled to be called at different time periods. The learning phase of the controller computes the service demands for the active VMs. Two possible states in learning phase indicate the request the initialisation of a learning phase and indication that a learning round was n course. After each round, a new QoS was computed and put it in XenStore database. The results show that their approach had met SLOs.

Another important paper [22] proposed a flexible distributed architecture for dynamic and autonomous management of a data centre equipped with autonomous VMs. The authors mentioned, the current VM migration can be achieved in several seconds or hours (numerous and independent) and can adjust resources in data centre dynamically and automatically. The resource management is usually carried out in a centralised manner by defining a utility function taking into account the levels of the resource utilisation. The proposed approach in this paper is based on a decentralised and independent VMs which in terms are not correlated to each other. They proposed a model formulation to efficiently solve the problem of dynamic allocation of interdependent VMs in virtual networks. They find the mapping between what is required as SLA and what is the availability of the resources in a data centre. These SLAs can be defined by several parameters such as response time, bandwidth, etc. The availability level of resources is the availability of CPU, memory, etc. In the first phase the model is used in a data centre and make the better second phase, as the levels of user requests in a data centre vary with time, it is important to predict these levels in advance for each VMs. The second phase aims to define a good network configuration of a data centre and calculates and applies VM migrations to set up a new configuration (must be flexible to be continuously updated with new SLAs criteria) which meets SLAs. The research results indicated that it is feasible to find a distributed and a decentralised VM allocation which promises flexibility and modularity for a good dynamic management of large-sized data centres. With virtual distributed resources, interdependent VMs can continuously exchange their data conveniently.

The researchers in the paper [23] proposed a distributed learning mechanism that facilitates self-adaptive VMs resource provisioning. They brought attention to the readers about the issue regarding the ability to partitioning hardware resource in VM instance to facilitate elastic computing environment to the users. They mentioned that this extra layer of resource virtualisation creates challenges when managing clouds effectively. The complicated relationship between co-hosted VMs and the arbitrary deployment of multi-tier application makes it difficult for administrators to plan good VM configurations. In IaaS computing, raw hardware infrastructure, such as CPU, memory and storage is provided to users as an on-demand virtual server. This ensures the proper utilisation of cloud resources. Cloud providers consolidate traditional web applications into a fewer number of physical servers to adjust the VM capacity. This brings the requirement of effective management of the resources to reconfigure each individual VMs in response to the change of application demands. The server virtualisation to optimise the performance resource utilisation works at an extent but VMs still have changes to interfere with each other and the practical issue such as VM's time delay to performance stabilisation after memory and CPU reconfiguration is a well-known fact. VM clusters of different users may overlap on physical servers. The overall VM deployment can show a dependent topology with respect to resources on physical hosts. VMs with mis-configuration can possibly become rogue ones affecting others. Furthermore, a mistake in the capacity management of one VM can spread onto the entire cloud. The authors of this paper tried to solve the issue by treating cloud resource allocation as a distributed learning task in which each VM being a highly autonomous agent which submits resource requests according to its own benefit. To handle workload dynamics, the extend VM configurations to VM running status and address the issues due to the use of continuous running status as the state space. More specifically, they solve the issue by treating VM resource allocation as a distributed learning task. Cloud users were given the ability to manage each VM capacity. Host agents evaluate the requests on one machine and send this as feedback to all the machines. VMs uses this feedback to learns its capacity management policy. The authors developed an efficient reinforcement learning approach for the management of individual VM capacity which is based on Cerebellar Model Articulation controller-based table. The resource efficiency was optimised by using a metric to measure a VM's capacity settings which synthesise application performance and resource utilisation. Their prototype implementation called iBaloon was able to find near-optimal configurations for a total number of 128 VMs in a 16 node closely correlated clusters with no more than 5% of performance overhead in a Xen-based cloud test-bed. The distributed approach is based on the scalability of infrastructure and highly rely on the implicit coordination between VMs belonging to the same virtual cluster. In this masters thesis project, our aim is to concentrate on VM sharing in a distributed manner rather than scalable approach, between the entire cloud infrastructure located around the globe rather than a cluster of VM in the same location approach.

2.6.4 Researches based on surveys

Last but not least, the following paper [24] need to be mentioned for its outstanding survey on various self-organising strategies for resource management in cloud computing. The paper not necessarily proposed a new idea or algorithm per se, but discussed and analysed various method of acquiring emergence behaviour of a distributed system. The authors of this paper discussed the techniques such as bio-inspired computing, multi-agent systems and evolutionary techniques to manage cloud components resources and how these solutions are applied in the management of these resources by the cloud providers. Among many papers, this research work draws our interest in adapting theory such as evolutionary game algorithm on this masters thesis project. The paper [24] mentioned that the system responsible for managing physical and virtual resources such as Resource Management System (RMS) have to aware of the current status of the infrastructure to determine whether there are enough resources available to satisfy incoming request. RMS monitors resources and fulfils the user requirements. Although the centralised approach is very common in RMS there lies a disadvantage of a single point of failure. In distributed RMS, many elements in the system share information for the decision making process. By doing so, these elements gain more autonomy and active participation in the overall system running. Distributed RMS minimises some of centralised approaches weaknesses, such as fault-tolerance and scalability. Techniques mentioned above are to make the resource more robust and adaptable. The authors describe that for decision making, a centralised manager can communicate with nodes to obtain current status and then determine what necessary steps to be taken to reach the optimised goal. In this type of system, the intelligence of the system as a whole is completely depended on the processing power and capabilities if the central node. It is easy to maintain or update this one node instead of each individual nodes but they are susceptible to failures, especially in dynamic networking. The best approach, in this case, would be distributed system, where we have the management function distributed across all nodes. The authors advocate that in order to achieve a equilibrium across all nodes or accomplish emergence behaviour an autonomic manager need to perform their activities based on a control loop called MAPE-K which monitors, analyse, plan, execute and gather knowledge about the infrastructure. Some of the bio-inspired solutions that are used to achieve such behaviour have very simple rules based on an idea that after several iterations the rules applied by organism lead to an emerging global behaviour. Problem-related to optimisation such as graph problems can be solved by applying ant colony optimisation (ACO). Multi-Dimensional Bin-Packing (MDBP) along with the ACO can be used to gain workload consolidation in which bins and the workload represent physical machine and the items to be packed respectively. A multi-objective ant colony for virtual machine allocation in Clouds can minimise total resource wastage and power consumption. Honey bees behave the similar to ants when foraging for foods. An overloaded virtual machine (like a bee without food) tries to schedule tasks to an under-loaded one (like a foraging bee finding a new source). A common social behaviour such as gossiping can be used as a conceptual solution in which nodes can exchange information with a set of other nodes, selected in a random way lead to a pairwise interconnection between them by means of active thread that is responsible for initiating the communication and a passive thread that accepts messages. Bio-inspired solutions are also a Multi-agent system (MAS) since they work based on agents to solve a given problem. MAS can be used for autonomic management of virtual networks in infrastructure to achieve self-healing of the networks. Agents in such system can implement an algorithm to select spare node of the physical infrastructure to associate with a node to minimise the link utilisation. The investigation provided by the researchers on self-organising strategies, their thorough insight and analytic approach to various related work inspired us to dive deep into this particular topic of research.

While researching for autonomous VM migration, it is observed that most work is done by concentrating on preferring either algorithms on improving distribution method or strategy and implementation to achieve autonomous VMs. Our project goal is to find a suitable algorithm to utilise the full aspect of a distribution method while balancing the autonomy and self-awareness of the VM in a network to acquire an optimal level of equality in resource sharing and most importantly possibility of VM migration without losing any downtime on the distributed web services.

Chapter 3

Methodology and Approaches

In traditional cloud computing, VMs are usually created in multiple regions to support the incoming requests that are coming from different geological locations towards the services that those VMs are providing. When one zone becomes busy or overloaded by the client's request, controller of those regional VMs are usually designed to spawn or scale up some more similar VMs to support the trade. The VMs in other zones can still be idle and stay the same as the controllers of those regional VMs have not got any further instructions to advertise their idleness to the different zones or any direction to help those over-loaded regions. This scalable framework works considerably until the region with heavy load runs out of resource pools and can become exhausted when every VMs in different zones gets topped out. Instead of scaling up to more VMs in affected (overloaded with incoming requests) zone, sharing of an exact amount of VMs for entire service globally while running the same service approach can be a better solution. It can reduce the cost of redundant usage of physical resources and can address the issue of the component limitations. As pointed out in the previous Chapter, the flexibility and independence of choosing to migrate to different affected regions by moving the decision making control from controller VMs to worker VMs itself seem more promising than traditional cloud computing setup. This equalises the incoming request or response time of the VMs over all the VMs in the same concurrent network. Our system design accommodates this methodology to solve the issue and adopt the self-organising VM approach to obtain maximise system resource utilisation.

3.1 Overview of the Methodology

In this section, the methodology to adopt evolutionary game theory for self-organisation, self-management and migration properties of the VMs along with the Erlang-c model (for calculating average regional response time) in the algorithm to archive an autonomous system is discussed. The algorithms that approach such method and implementation of the system design on the VMs are discussed in the latter sections.

Before we discuss the methodology of our system design, a brief description of the collectivity of animal behaviour is discussed here. A collective animal behaviour is a form of behaviour which involves the coordinated behaviour of the large groups of similar animals. These groups also show the emergent properties among them. The concept of self-organisation has been used to understand collective animal behaviours of the animals. The simple and repeated interactions between individuals can produce complex adaptive patterns at the level of the group. The behaviour includes the costs and benefits of group's memberships, decision-making process, locomotion and synchronisation and transfer of information across the group. Studying the principles of such collective behaviour has relevance to computer engineering problems through the philosophy of biomimetics (imitation of the models). Inspiration coming from patterns in physical systems, such as spiral chemical waves, starling flocks effects, dance model of bees, etc. arise without complexity at the level of the individual units of which the system is composed as a whole.

Unlike the simple units composing physical systems, however, animals are themselves complex entities [25]. Using the same principle in computer science, especially in cloud computing the system process can be adapted to the same pattern and emerge as an infrastructure. It is important to know the internal structure of the group because that structure can be related to the proposed motivations for animal grouping. Once the location of each animal at each point in time is known, various parameters describing the animal group can be extracted. One of the good analogy was given in the article [26] by the biologist David Harper, where he conducted some experiments on ducks in the ponds to University Botanic Garden of Cambridge in the winter of 1979-1980. The experiments give us a better understanding of how equilibria can be achieved by repeated iterations carried by decision-making rules. With the flock of 33 ducks, Harper places two observers in two fixed points of the lake surface 20 meters apart. The observers throw pieces of bread in regular intervals at different distribution frequency of bread pieces per minute. After a certain number of iterations and experiments, it was shown that the after about a minute or two the number of ducks in the least profitable point (fewer pieces of bread per minute) was stabilised to 11 out of 33. On the point where more pieces of bread are per minute were thrown had more ducks (rest 22 ducks). The experiments also informed us that, it was observed that at any point if a duck leave its point will get less amount of bread. At the end of the experiment, the number of ducks were stabilised to a certain amount at both points and all the ducks in both points get an almost exact amount of piece of bread. At the beginning of the experiment, most ducks behaved as optimisers and went to the point where the higher amount of bread per minutes are being tossed. The site or point selection of the ducks are then considered as not just an optimisation problem but more like a game. A single ducks choice no only influence the outcome but also the choice of other ducks. Ducks were together reached to a equilibrium as a whole group where they did not know the game they were playing. The final results proved that a emergence can result if every entity in the evolutionary game plays by the players with the similar entity, i.e., same rule, behave exactly the same and takes the repeated action in a similar manner individually and independently.

In this project, the behavioural algorithms of collectivity in nature are identified and implemented in the system design. In the cloud platform, the VMs can be recognised as animals when the incoming requests to each zone can be referred as food/water. As a group of VM, the equalisation of the average response time of each zone by sharing the amount of VMs between affected zones can be recognised as the collective animal behaviour pattern. When the number of incoming request gets higher in a zone with few numbers of VMs, due to the high demand for web service access, the average response time on that individual zone can become scarce. The total response time of each zone become overloaded when the service rate becomes lower than the request rate. This can mostly happen due to the geographical location of web service requester at a particular time. The higher the number of the web request the higher the request rate becomes. This requires an increase in the service rate which can be simply done by spawning more VMs at that effective zones. The scaling is effective as long the idle systems are not underutilised which is a common issue in virtualisation systems. However, in the cloud platform scaling can become an issue, when the service providers demand an amount in cost for each running VM. When the demand for web services increases it becomes important to ensure full utilisation of all the idle and running VMs before deploying new ones. Based on the geological location of the VMs, it is possible that some VMs are underutilised due to a fewer web service request in that region. To avoid that, an algorithm is proposed in this masters thesis project, by using the emergence principal of animal group, to optimise the idle VM sharing among all the zones to equalise the total average response time of the whole framework.

3.1.1 Evolutionary game theory

Evolutionary game theory (EGT) is the application of game theory to evolving populations in biology. Darwinian competition can be modelled by defining a framework of contests, strategies, and analytic. It originated in 1973 with J. M. Smith and G. R. Price's formalisation of contests, analysed as strategies, and the mathematical criteria that can be used to predict the results of competing strategies [27]. EGT is different from the classical game theory where the dynamics of strategy changes are more focused and the frequency of the competing strategies in the population are taken into account. This theory has helped to explain several basis of altruistic behaviours in Darwinian evolution as well in the economy, sociology and needless to say in computer science. The author of article [27] realised that an evolutionary version of game theory does not require players to act rationally but have a strategy on its own. The tests with EGT strategies shows that the results for the ability to survive and reproduce. In biology, generally, the strategies which are genetically inherited that controls an individual's action. This is analogous to computer programs. The success of a strategy results by how good the strategy is in the presences of competing strategies and the frequency with which those strategies are used [28]. Participants of this games aim to reproduce as many replicas of themselves as they can and the pay-off is in units of fitness. Rules include replicator dynamics or fit players will spawn more replicas themselves into the population and how the less fit will be picked out in replicator equation which involves passing on of genetic traits from parents to their offspring but no mutations. Games as such run repetitively with no terminating conditions until the strategies succeeded and any equilibrium has been reached.

Although the evolutionary game theory was originally developed for biology, its applications in other fields are growing due to the following reasons [12]:

- **Solution refinement:** In traditional game theory, the Nash equilibrium (each player is assumed to know the equilibrium strategies of the other players) solution approach ensures that a player cannot improve its pay-off if none of the other players in the game deviates from the solution. However, when the solution to a non-cooperative game has multiple Nash equilibria, a refined solution is required. Evolutionary equilibrium, which is based on the theory of evolutionary game theory, provides stability, i.e., a group of players will not change their chosen strategies over time.

- **Bounded rationality:** Unlike a classical single-play non-cooperative game, in which all of the players make decisions that lead immediately to the desired solution, an evolutionary game involves players slowly changing their strategies to achieve the solution eventually.

- **Dynamics in the game model:** An evolutionary game can explicitly capture the dynamics of interaction among the players in a population. In an evolutionary game, learn from the observations, and make the best decision based on its knowledge. In addition, with replicator dynamics, the state of the game can be determined at a particular point in time, which is useful for investigating the trajectory (i.e., trend) of the strategies of the players while adopting their behaviour to reach the solution.

In this project design, the concept of EGT is adapted to design a strategy for all the participants in the complex system, i.e., VMs in the cloud. The design is to aim an equilibrium among the average response time and to achieve QoS of entire distributed web service provided by those VMs. Using this theory, the VMs aim to equalise the total average response time in global infrastructure to show an emergence behaviour as the whole system. The design works as when the number of the response time of one region escalates over certain level due to the factors of the increased number of incoming request to access these web services and VM managing those request becomes lower than a certain optimal number. Then the other VMs from a different region, which handles the same web service request, takes account of certain scenario and equalises the response time of affected sites by sharing their own idle VMs considering that helping region have a better response time than all the regions. In this scenario, every VM can individually decide what should be its next move (stay in the region or help the affected region). The algorithm design will be explained thoroughly in the latter sections.

3.1.2 Erlang Unit

The Erlang is a dimensionless unit especially used in telephone as a measure of offered load or carried the load on service-providing elements such as telephone circuits or switching equipment. Full utilisation of circuit capacity constitutes a single Erlang [29]. When used in carried traffic, a non-integer usually represents the average number of concurrent calls carried by the service-providing elements. This average is calculated over some reasonable period of time. One Erlang of carried traffic refers to a single refers to the continuous use or two channels are being used 50% of the time. When used to describe the offered traffic represents the average number of concurrent calls that would have been carried if there were an unlimited number of circuits, considering none of the calls has been rejected. The relationship between these offered carried traffic depends on the design of system and user behaviour.

Offered traffic (in Erlang) is related to the call arrival rate, λ, and the average call-holding time (the average time of a phone call), h, by: $E = \lambda h$, where λ(the mean arrival rate of new calls), h (the mean call length or holding time)and E (the traffic in Erlang) using the same units of time (seconds and calls per second, or minutes and calls per minute). Three common Erlang models are: callers whose call-attempts are rejected go away and never come back, callers whose call-attempts are rejected try again within a fairly short space of time, and the system allows users to wait in the queue until a circuit becomes available. Erlang-B and Erlang-C formula is the most commonly implemented formulas at the present.

Erlang-C formula

In this project, among all the Erlang formulas, Erlang-C formula is adopted. The Erlang-C formula defines the probability that an arriving customer or incoming web access request will need to queue instead of immediately being served [30]. The formula includes a queue property in all the systems. The formula assumes an infinite population of sources which offer traffic of an Erlang to N server together. However, if all the servers are busy when a request arrives from a source, the request gets queued. The queue can hold an unlimited number of requests simultaneously. The Erlang-c formula calculates the probability of queuing offered traffic, assuming that blocked requests stay in the system until they can be handled. The formula assumes the request will never get dropped and increasing the number of agents (in our case VMs) will maintain the desired service level. The mathematical expression of Erlang-C formula can be expressed as follows:

$$P_W = \frac{\frac{A^N}{N!}\frac{N}{N-A}}{(\sum_{i=0}^{N-1}\frac{A^i}{i!}) + \frac{A^N}{N!}\frac{N}{N-A}}$$

Here: A is the total traffic offered in units of Erlangs, N is the number of servers and P_W is the probability that a customer has to wait for service. The call arrivals can be modelled by a Poisson process and that call holding times are described by a negative exponential distribution.

Limitations of the Erlang formula

When Erlang developed Erlang-B and Erlang-C traffic equations, they were developed based on a set of assumptions, which are accurate in most situations, especially when there is high traffic congestion. But these formulas fail to predict accurately the correct number of agents required because of re-entrant traffic. At peak times, this leads the congestion of high-traffic to more congestions. Additional circuits or agents needed to be available at that certain congested traffic period in order to avoid this high-loss of traffic. There are some rules to establish a queuing with Erlang such as [1]:

- If the number of jobs becomes infinite, the system becomes unstable. For stability, have to make sure that the mean arrival rate is always less than mean service rate. i.e.,
$$\lambda(mean\ arrival\ rate) < m\mu(mean\ service\ rate\ per\ server)$$
- Finite population cannot have an infinite queue and finite queue drops if too many arrive. So never has infinite queue
- Number of jobs is equal to waiting and servicing, i.e.,
$$n = n_{queue} + n_{service}$$
or, number of jobs in system = number of jobs waiting in queue + number of jobs receiving service
- If jobs not lost due to buffer overflow the mean jobs is related to response time as:
$$mean_{(jobs\ in\ system)} = arrival\ rate * mean_{(response\ time)}$$
This also known as Little's law [31].
- Time spent in system is sum of queue and service time, i.e.,
$$r(response\ time) = w(waiting\ time) + s(service\ time\ per\ job)$$

3.1.3 Migration of Virtual Machines

The term migration of VM refers to the moving of a machine entirely as a unit from one physical server to another, in cloud term from one regional location to another. Migration can be distinguished as two types: live and cold migration. In the live migration, there is no downtime, that capability opens up a variety of practical uses. Cold migration, however, is the migration technique when the VMs are not required to be an shared storage during migration. But it does demand a shut-down of the VM before it migrates. CPU compatibility checks do not apply when migrating a VM with cold migration. Some of the main tasks of cold migration are if the different data store is chosen as the migrating destination, the configuration files such as NVRAM files (BIOS), virtual disks (if chosen) and log files are transferred from source to destination hosts storage area. After the completion of migration, the older version of VM is deleted from the source host. Fig. 3.1 shows the traditional model for VM migration between different servers in the cloud environment.

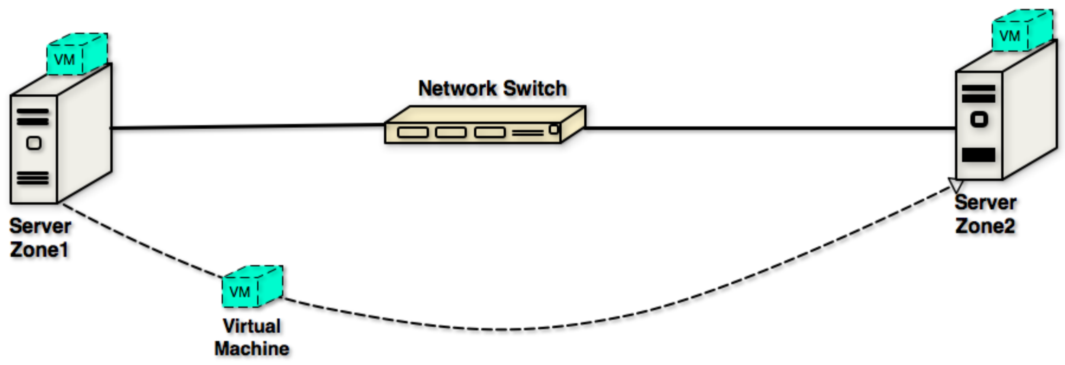


Figure 3.1: Illustration of Traditional Model for Virtual Machine Migration

3.1.4 Self-management

The growing complexity of modern networked computer systems is currently the biggest limiting factor in their expansion. In order to manage themselves, the autonomic systems need to observe situations and events, "sense" their environment, and then make decisions about which actions to execute and when. The aim of the system is to define a behavioural schema which can be categorised as a different function of the essence of the autonomic systems in self-management. Some properties of self-management of systems for this project's system design should be as follows [24]:

- The system should be able to reconfigure itself to handle changes in its environment or requirements without any human intervention.
- The system should be able to automatically adjust itself and related components to handle common and frequent events, e.g., adding and removing nodes.
- Self-configuration properties need to be introduced, which means service architecture will continue to work when nodes are added or removed during execution. When a new node is introduced into an autonomic system, it will automatically learn about its environment and then integrate itself. Meanwhile, if a node is removed, the other nodes will also be aware of the changes and they can modify their own behaviour to adopt the new situation.
- Self-healing properties need to be implemented, which will not rely on manual interaction for identifying and debugging failures. An autonomic system can detect, diagnose, repair, and sometimes predict the problems and failures on its own, regardless of the origin and nature of the problem. The purpose of self-healing is to pull the system back from the wrong states into the desired states. This kind of behaviour is called convergence. If a system is fully convergent, whatever the initial state the system has, it will manage itself back to the equilibrium state.
- Self-optimisation properties can be implemented, which ensure after an extended period of gaining "experience", an autonomic system will be able to learn, and then continuously evaluate and change its run-time parameters to improve its operation. Experience and log memory can be kept as a knowledge base the systems to use it to find, verify and apply appropriate changes to upgrade their functionality.

3.2 Approaches

In order to achieve the project goal, an algorithm to adopt Erlang-C formula with EGT has been proposed in this masters thesis. This simple but effective algorithm takes into account the total average response time of each zone where the number of VMs are responding to web services requests. These response time are gathered and learned by each individual VMs and a decision has been made in order to aim for equalisation of response time throughout the entire infrastructure of the distributed system. In this way, the convergence of the whole system as one unit is reached where each individual VM migrate to different affected zones and balance the requirements of service agents. Different variants of the algorithm have been proposed to evaluate the system performance as an autonomous system.

3.2.1 General algorithm

The algorithm proposed in this project uses an evolutionary game theory approach. The general principles of the algorithm are as follows:

- In this algorithm, each VM has its own ability to choose between either to stay in the current zone or move to an affected zone. Here affected zone refers to the zone which has an average response time less than a certain threshold.

- Each VM respond to web service requests from clients which are distributed via a load balancer.
- Every individual VMs can calculate its own zone's average response time from the incoming request rate, service rate and number of VMs of that zone. This can be calculated with Erlang-C model for multiple queues with the following equation:

$$\rho = \frac{\lambda}{\mu * c} \tag{3.1}$$

  where, ρ = Utilisation, λ = Request Rate, μ = Service rate and c = Number of Servers, followed by Erlang-C formula (k) if more than c amount of jobs,

$$\begin{aligned} P_r(> c\ jobs) &= \rho_c + \rho_{c+1} + \rho_{c+2} + \dots \\ &= \sum_{c+1}^{\infty} P_n \\ &= \frac{P_0(cp)^c}{c!} * \sum \rho^{n-c} \\ &= \frac{[(c\rho)c]}{[c!(1-\rho)]} P_0 \end{aligned} \tag{3.2}$$

- The average response time is calculated with the following:
  - Probability when a packet comes, it needs to queue in the buffer. That is, P(W>0) = 1 - P(N < c), Also known as Erlang-C function => Prob. Queue
  - Average time of packets spending in the queue => Average Queue Time
  - Return the average time of packets spending in the system (in service and in the queue). i.e., (Prob. Queue / Average Queue Time) + (1.0 / service)
- Next the VM calculates the total average response time of all the zones in the infrastructure.
- Based on the variants (Sec. 3.2.2 - 3.2.5) and conditions apply (such as VM's current zone's response time is under the average threshold, different zone's condition and the probability of moving), VM decides to move to a candidate (affected) zone which matches the certain criteria. The probability represents the chances of each VM to each zone in percentage. It also represents how much better the average response time of the current zone of the VM is compared to other zones. If no condition matches VM stays in the current zone.
- VM continues to respond to the incoming requests until it calculates the moving probability again in a certain interval.
- After a number of iterations, the average response time of all the participating zones becomes equalised.

The proposed evolution algorithm is as follows:

Algorithm 1 Evolution Algorithm

```
1: Suppose, we have $N$ number of zones denoted as $Z$, where $Z = \{Z_1, Z_2, Z_3, \ldots, Z_N\}$.
2: Suppose, we have $S$ number of VMs denoted as $VM$, where $VM = \{VM_1, VM_2, VM_3, \ldots, VM_S\}$.
3: Let, $r_i(t)$ denote the current response time at zone $Z_i$.
4: Let, $l_{ij} = 1$ if $VM_i$ is in $Z_j$, otherwise 0. It gives the location of $VM_i$ at instant time $t$.
5: At time $t_i$, $VM_i$ collects average response time of current zone $r_i(t)$, according to
   equation (3.2).
6: $VM_i$ calculates total average response time of $Z_N$ derived by $\bar{r}(t) = \frac{\sum_{i=1}^{N} r_i(t)}{N}$ and checks
   if $r_i(t) < \bar{r}(t)$.
7: If true $VM_i$ picks a candidate zone using one of the variants of algorithm, satisfying
   $r'_i > r_i$ where $i' \neq i$.
8: $VM_i$ will move to a candidate zone with the probability $\frac{\bar{r}-r_i}{\bar{r}}$.
9: Repeat from Step 5 to Step 8.
```

Following up on the above general algorithm different variants of the proposed algorithm are discussed in the following sections.

3.2.2 Uniform-site migration (naive)

In Naive Uniform-Site migration algorithm, the VM selects any zone naively as the candidate zone to move. This is done in step 7 of the general evolution Alg. (1). The candidate zone does not necessarily have the less average response time than the threshold of the average response time of all the zones. According to this migration, the VM moves to any zone ones it decides to migrate. Hence the name naively. This algorithm is designed to experiment the behaviour of a VM in a zone taking the decision of next steps without any having any knowledge or information of the network condition it belongs to. This results the number of VMs in the affected zones fluctuates due to the fact is that none of the VM deciding to migrate to equalise the average response time of the zones, rather it migrates anywhere even the response time is in much better condition than other concurrent zones. Naiveness of in the VM as a character to make a migration decision makes the entire network converging in headless or aimless manner.

The algorithm works as follows:

Algorithm 2 Uniform-Site Migration (Naive) Algorithm

```
1: Suppose, we have $N$ number of zones denoted as $Z$, where $Z = \{Z_1, Z_2, Z_3, \ldots, Z_N\}$.
2: Suppose, we have $S$ number of VMs denoted as $VM$, where
   $VM = \{VM_1, VM_2, VM_3, \ldots, VM_S\}$.
3: Let, $r_i(t)$ denote the current response time at zone $Z_i$.
4: Let, $l_{ij} = 1$ if $VM_i$ is in $Z_j$, otherwise 0. It gives the location of $VM_i$ at instant time $t$.
5: At time $t_i$, $VM_i$ collects average response time of current zone $r_i(t)$, according to
   equation (3.2).
6: $VM_i$ calculates total average response time of $Z_N$ derived by $\bar{r}(t) = \frac{\sum_{i=1}^{N} r_i(t)}{N}$ and
   checks if $r_i(t) < \bar{r}(t)$.
7: If $r_i(t) < \bar{r}(t)$ in Alg. (1) is true, $VM_i$ choose $Z_i$ randomly, where $Z > 0$.
8: $VM_i$ will move to candidate zone with the probability $\frac{\bar{r}-r_i}{\bar{r}}$.
9: Repeat from Step 5 to Step 8.
```

The figure (Fig. 3.2) illustrates the flow diagram of Uniform-Site Migration.

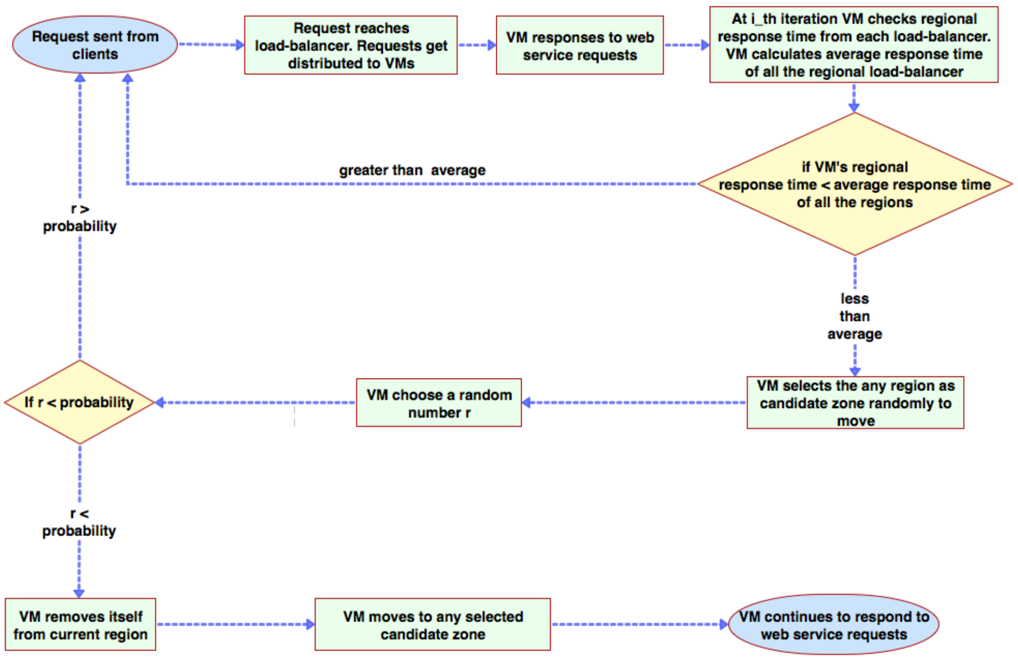

Request sent from clients
Request reaches load-balancer. Requests get distributed to VMs
VM responses to web service requests
At i_th iteration VM checks regional response time from each load-balancer. VM calculates average response time of all the regional load-balancer
if VM's regional response time < average response time of all the regions
greater than average
less than average
VM selects the any region as candidate zone randomly to move
VM choose a random number r
If r < probability
r > probability
r < probability
VM removes itself from current region
VM moves to any selected candidate zone
VM continues to respond to web service requests

Figure 3.2: Flow-Diagram of Uniform-Site Migration (Naive)

3.2.3 Uniform-site migration (informed)

In the Informed Uniform-Site migration algorithm the VM selects the candidate zone to move based on which zone's response time is worse than the VM's current zone's response time. This is done in step 7 of the general evolution Alg. (1). If the potential candidate zone > 1, VM picks a random candidate zone. In this algorithm, the idea of emergence behaviour has been reasonably implemented. Using this algorithm VM can independently takes the next migration steps (which zone to choose as candidate zone to migrate to) with some information about the corresponding zone's information. The goal of this migration technique not to resolve the issue with high average response time in a single step but gradually come to the convergence point slowly. The theory evolutionary game has been introduced in this migration scheme where the repeated rational decision making of individual entity (in this case VMs) will result in a system convergence as a whole system. The migrating VM will not choose the most affected zone as candidate automatically, rather choose randomly any other zones which are affected more than its currently geo-located zone. This means the least affected zone in the infrastructure will never be chosen as the candidate zone. Number of VMs in a zone with high traffic (or multiple zones with lesser traffic) will eventually get distributed amongst all the zone eventually.

The algorithm works as follows:

Algorithm 3 Uniform-SIte Migration (Informed) Algorithm

```
1: Suppose, we have $N$ number of zones denoted as $Z$, where $Z = \{Z_1, Z_2, Z_3, \ldots, Z_N\}$.
2: Suppose, we have $S$ number of VMs denoted as $VM$, where
   $VM = \{VM_1, VM_2, VM_3, \ldots, VM_S\}$.
3: Let, $r_i(t)$ denote the current response time at zone $Z_i$.
4: Let, $l_{ij} = 1$ if $VM_i$ is in $Z_j$, otherwise 0. It gives the location of $VM_i$ at instant time $t$.
5: At time $t_i$, $VM_i$ collects average response time of current zone $r_i(t)$, according to
   equation (3.2).
6: $VM_i$ calculates total average response time of $Z_N$ derived by $\bar{r}(t) = \frac{\sum_{i=1}^{N} r_i(t)}{N}$ and
   checks if $r_i(t) < \bar{r}(t)$.
7: Suppose, we have $M$ number of candidate zone $Z^m$, where
   $Z^m = \{Z_1^m, Z_2^m, Z_3^m, \ldots, Z_M^m\}$.
8: If $r_i(t) < \bar{r}(t)$ in Alg. (1) is true, $VM_i$ choose $Z_i^m$ randomly, where $Z^m >= 0$.
9: $VM_i$ will move to candidate zone with the probability $\frac{\bar{r} - r_i}{\bar{r}}$.
10: Repeat from Step 5 to Step 9.
```

The figure (Fig. 3.3) illustrates the flow diagram of Uniform-Site Migration.

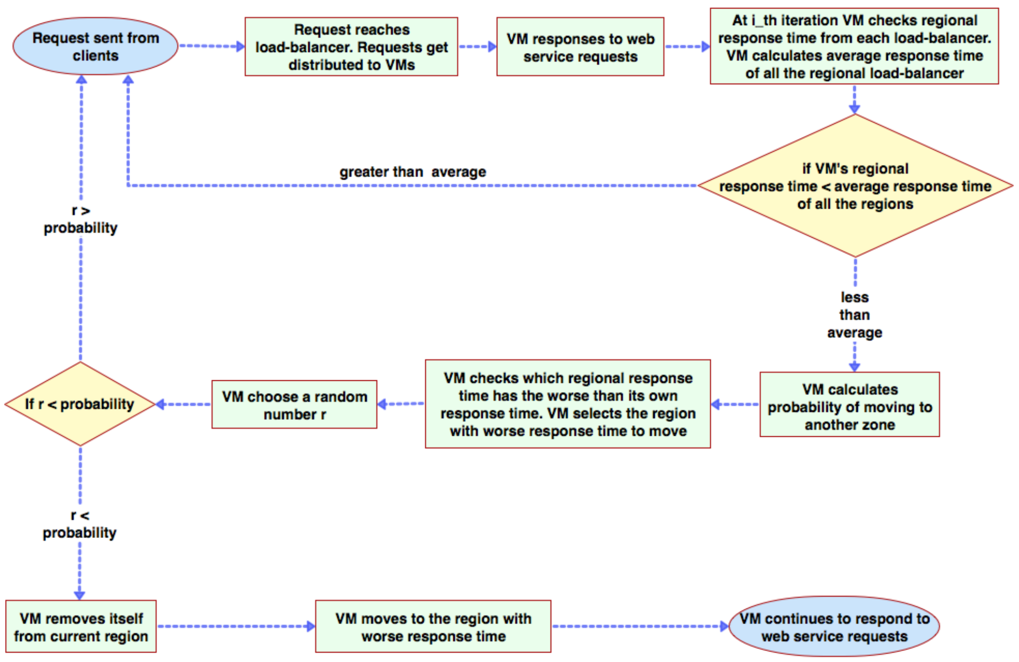

Request sent from clients
Request reaches load-balancer. Requests get distributed to VMs
VM responses to web service requests
At i_th iteration VM checks regional response time from each load-balancer. VM calculates average response time of all the regional load-balancer
if VM's regional response time < average response time of all the regions
greater than average
less than average
VM calculates probability of moving to another zone
VM checks which regional response time has the worse than its own response time. VM selects the region with worse response time to move
VM choose a random number r
If r < probability
r > probability
r < probability
VM removes itself from current region
VM moves to the region with worse response time
VM continues to respond to web service requests

Figure 3.3: Flow-Diagram of Uniform-Site Migration (Informed)

3.2.4 Biased migration

The VM with this migration scheme calculates the candidate zone to move/migrate based on the probability for each zones. Zones with the highest probability get the priority as a candidate zone. VM picks the candidate zone with the highest probability to move. At a particular moment, the migrating VM (the VM that has decided to migrate to a different zone) can calculate the probability to pick the candidate zone based on the average response time of that each connected zone in the network. In this case, the decision made by moving VM is relatively affected by not just by the individual zones performance but all the zones in the network as VM calculates the probability by calculating the total average of all zones average response time including its own. This scheme is designed to influence entire network condition of the framework as a single system. It is a slight upgrade from a previous scheme (Alg. 3) where the VMs are given a better decision making properties to observe the adjoining neighbours condition. This algorithm works as follows:

Algorithm 4 Biased Migration Algorithm

```
1: Suppose, we have $N$ number of zones denoted as $Z$, where $Z = \{Z_1, Z_2, Z_3, \ldots, Z_N\}$.
2: Suppose, we have $S$ number of VMs denoted as $VM$, where
   $VM = \{VM_1, VM_2, VM_3, \ldots, VM_S\}$.
3: Let, $r_i(t)$ denote the current response time at zone $Z_i$.
4: Let, $l_{ij} = 1$ if $VM_i$ is in $Z_j$, otherwise 0. It gives the location of $VM_i$ at instant time $t$.
5: At time $t_i$, $VM_i$ collects average response time of current zone $r_i(t)$, according to
   equation (3.2).
6: $VM_i$ calculates total average response time of $Z_N$ derived by $\bar{r}(t) = \frac{\sum_{i=1}^{N} r_i(t)}{N}$ and
   checks if $r_i(t) < \bar{r}(t)$
7: Suppose, we have $M$ number of candidate zone $Z^m$, where
   $Z^m = \{Z^m_1, Z^m_2, Z^m_3, \ldots, Z^m_M\}$.
8: Let, $prob\_Z^m_i$ denote the probability to move at candidate zone $Z^m_i$.
9: $VM_i$ will pick and move to the candidate zone with $max(prob\_Z^m_M)$ considering
   $r_i(t) < \bar{r}(t)$ in Alg. (1) is true.
10: Repeat from Step 5 to Step 9.
```

The figure (Fig. 3.4) illustrates the flow diagram of Biased Migration.

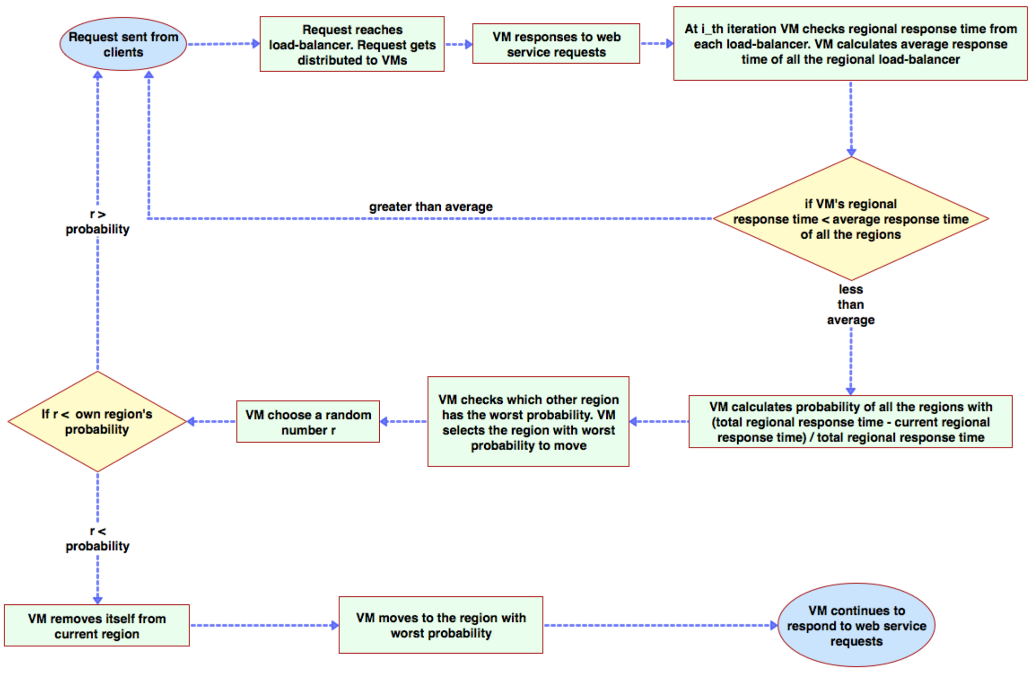

Request sent from clients
Request reaches load-balancer. Request gets distributed to VMs
VM responses to web service requests
At i_th iteration VM checks regional response time from each load-balancer. VM calculates average response time of all the regional load-balancer
if VM's regional response time < average response time of all the regions
greater than average
less than average
VM calculates probability of all the regions with (total regional response time - current regional response time) / total regional response time
VM checks which other region has the worst probability. VM selects the region with worst probability to move
VM choose a random number r
If r < own region's probability
r > probability
r < probability
VM removes itself from current region
VM moves to the region with worst probability
VM continues to respond to web service requests

Figure 3.4: Flow-Diagram of Biased Migration

3.2.5 Single-point migration

In this Single-Point Migration algorithm, the VM selects the candidate zone to move based on which zone's response time is worse than all zone's average response times. This is done in step 7 of the general evolution Alg. (1). VM picks the zone with the highest average response time i.e., the zone which is currently affected most, as the candidate zone. The single-point reference for this migration naming coming from the concept where the migrating VM tends to move to the same zone in a specific point of time. The designing of this algorithm targeted to converge the distributed web service structure faster since all the migrating VM choose the same zone as their destination. The main goal is to meet the demand of most resource-deprived zone before the other less affected zones. The algorithm works as follows:

Algorithm 5 Single-Point Migration Algorithm

```
1: Suppose, we have $N$ number of zones denoted as $Z$, where $Z = \{Z_1, Z_2, Z_3, \dots, Z_N\}$.
2: Suppose, we have $S$ number of VMs denoted as $VM$, where
   $VM = \{VM_1, VM_2, VM_3, \dots, VM_S\}$.
3: Let, $r_i(t)$ denote the current response time at zone $Z_i$.
4: Let, $l_{ij} = 1$ if $VM_i$ is in $Z_j$, otherwise 0. It gives the location of $VM_i$ at instant time $t$.
5: At time $t_i$, $VM_i$ collects average response time of current zone $r_i(t)$, according to
   equation (3.2).
6: $VM_i$ calculates total average response time of $Z_N$ derived by $\bar{r}(t) = \frac{\sum_{i=1}^{N} r_i(t)}{N}$ and
   checks if $r_i(t) < \bar{r}(t)$.
7: Suppose, we have $M$ number of candidate zone $Z^m$, where
   $Z^m = \{Z_1^m, Z_2^m, Z_3^m, \dots, Z_M^m\}$.
8: If $r_i(t) < \bar{r}(t)$ in Alg. (1) is true, $VM_i$ choose $max(Z_N Z_M^m)$ as candidate zone, where
   $Z^m >= 0$.
9: $VM_i$ will move to the candidate zone with the probability $\frac{\bar{r}-r_i}{\bar{r}}$.
10: Repeat from Step 5 to Step 9.
```

The figure (Fig. 3.5) illustrates the flow diagram of the algorithm.

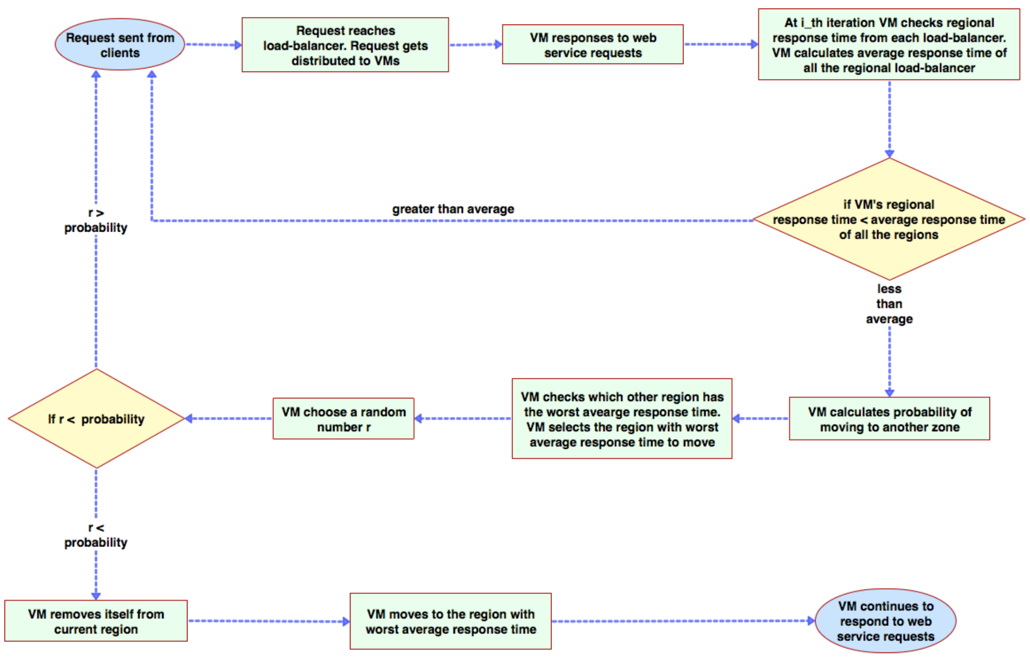

Request sent from clients
Request reaches load-balancer. Request gets distributed to VMs
VM responses to web service requests
At i_th iteration VM checks regional response time from each load-balancer. VM calculates average response time of all the regional load-balancer
if VM's regional response time < average response time of all the regions
greater than average
less than average
r > probability
VM calculates probability of moving to another zone
VM checks which other region has the worst avearge response time. VM selects the region with worst average response time to move
VM choose a random number r
If r < probability
r < probability
VM removes itself from current region
VM moves to the region with worst average response time
VM continues to respond to web service requests

Figure 3.5: Flow-Diagram of Single-Point Migration

3.2.6 Uniform-site migration (informed) using peer-to-peer communication

This variant is slightly different than previous variants. This variant is not an additional scheme rather considered as an extension to the previous ones. In this uniform-site migration algorithm, the zones are separated into groups, where each group has can have multiple zones. The groups represent the connected zones as each group will try to equalise their average response time over a period of time. Each group/cluster can have at least one zone in common with another group or can be completely unaware of the other clusters. This ensures all the zones are part of one or more groups in a topology. Fig. 3.6 shows the number of groups with sets of VMs where each group have at least one zone in common, i.e., while a group try to balance the average response time among the zones it contains can be also affecting the zone from different cluster or group. Each group equalises the response time using the method in Alg. 3. VM moves to the candidate zone within the same group or cluster and the communication between inter-connected clusters are considered to be peer-to-peer. If a common zone between two clusters is not affected by the evolutionary steps of any of the clusters, the other connected cluster can remain idle through the entire period.

The aim of this variant is to examine the performance of a uniform-site migration (informed) scheme in different topology scenario. The migration variant is based on not just the migration principal of the VM but also the inter-relationship with the other connected zones and the network condition of those zones regarding their connectivity. The scheme tends to be broadened to take cluster based connectivity (where the zones can be part of one or multiple clusters) into account. Many web services can be designed to be separated in multiple clusters depending on there geological distance and area of serving. In this scenario, not every zone are considered to be aware of the network status of all the zones but only their associative zones in the regional data-centre. Given the scenario, the emergence of a system not only can affect the performance of the own conjoint member (zone) of the same cluster, but also the zones those members might be affiliated with. It can also possible that the affected zone in the cluster can never affect the number of VMs in common zones between multiple clusters. This is to prove the concept that an action of a single VM in a zone from the separate cluster can impact the outcome of the data-centre in a completely different cluster which can be also not directly inter-linked together. Migrating VMs will never move to another cluster even though there is a common zone between them.

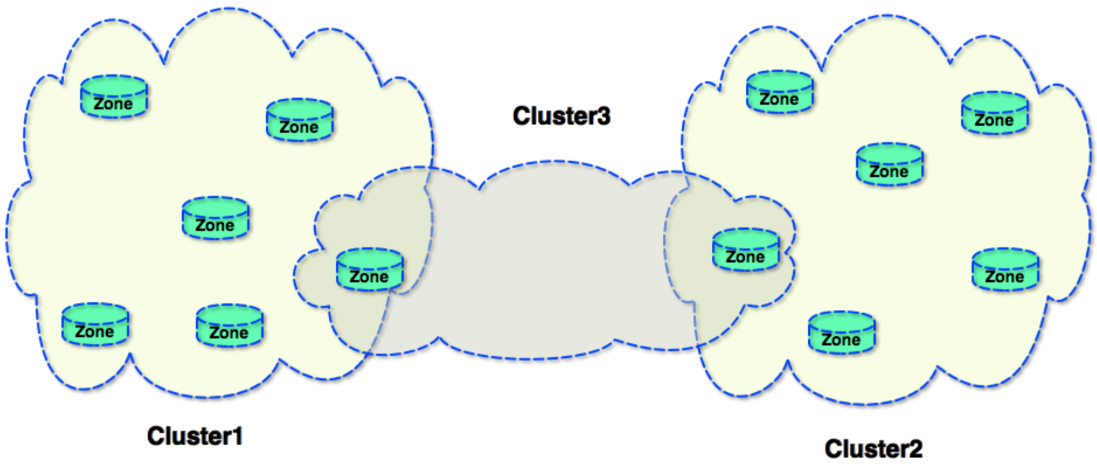


Figure 3.6: Zones Separated in Multiple Clusters

The algorithm works as follows:

Algorithm 6 Peer-to-Peer Communication Algorithm

```
1: Suppose, we have $N$ number of zones denoted as $Z$ in each group, where
   $Z = \{Z_1, Z_2, Z_3, \dots, Z_N\}$.
2: Suppose, we have $S$ number of VMs denoted as $VM$ in each zone, where
   $VM = \{VM_1, VM_2, VM_3, \dots, VM_S\}$.
3: Suppose, we have $X$ number of groups denoted as $G$, where
   $G = \{G_1, G_2, G_3, \dots, G_X\}$. Each $G_i$ have a set of two or more zones.
4: Let, $r_i(t)$ denote the current response time at zone $Z_i$.
5: Let $G_i r_i(t)$ denotes the current total average response time at $G_i$.
6: Let $G_i l_{ij} = 1$ if $VM_i$ is in $Z_j$ of $G_i$, otherwise 0. It gives the location of $VM_i$ at
   instant time $t$.
7: At time $t_i$, $VM_i$ collects average response time of current zone $r_i(t)$, according to
   equation (3.2).
8: $VM_i$ calculates total average response time of $G_i Z_N$ derived by $G_i \bar{r}(t) = \frac{\sum_{i=1}^{N} G_i r_i(t)}{N}$
   and checks if $G_i r_i(t) < G_i \bar{r}(t)$.
9: If true $VM_i$ finds $max(G_i Z_N \bar{r}_i)$ and picks it as candidate zone to move.
10: $VM_i$ will move to candidate zone with the probability $\frac{\bar{G_i r} - r_i}{\bar{G_i r}}$.
11: Repeat from Step 7 to Step 10.
```

The figure (Fig. 3.7) illustrates the flow-diagram of variant Uniform-Site Migration using Peer-to-Peer communication.

3.2.7 Uniform-site migration (informed) using graph partitioning

In this algorithm, Each VMs are part of one or two groups same as Alg. 6. This variant is also an extension of Alg. 3, similar to peer-to-peer communication variant. The distinction in this variant is that the VMs are bidirectional and VM in all the common zones in multiple groups consider all the conjoined zones and their average response time while deciding to migrate to affected zones. VMs can migrate to a different group as long as the zones they are belongs to are part of that group. All the groups are connected in a ring topology as shown in Fig. 3.8. This partitioning scenario is designed to implement the variant of Alg. 1, where the main focus is to investigate the performance of proposed evolution algorithm. The VMs are all part of a single zone which must be connected to at least two different zones. The scheme works in such a way that migration of a VM in a zone can affect only the zones that are directly linked to the group they are part of. Implementing the informed uniform-site migration in this scheme of general algorithm, each VM only has the choice to migrate into a candidate zone which is in either of two groups. Partitioning the zones in several different groups can/might interfere with the convergence behaviour of the distinct group, i.e., a particular group of two connected zones only. It is a different approach to peer-to-peer communication variant and this scheme is the extended version of the first four basic variants of the evolutionary algorithm. The variant is to examine the performance of a system, where the evolutionary game theory can only be applied to a single partition when at the same time other co-related zones from different partitions can be affected by the evolutionary procedure.

The Graph Partitioning variant of the algorithm works almost the same as Alg. 7 except step 8 and 9 which is as follows:

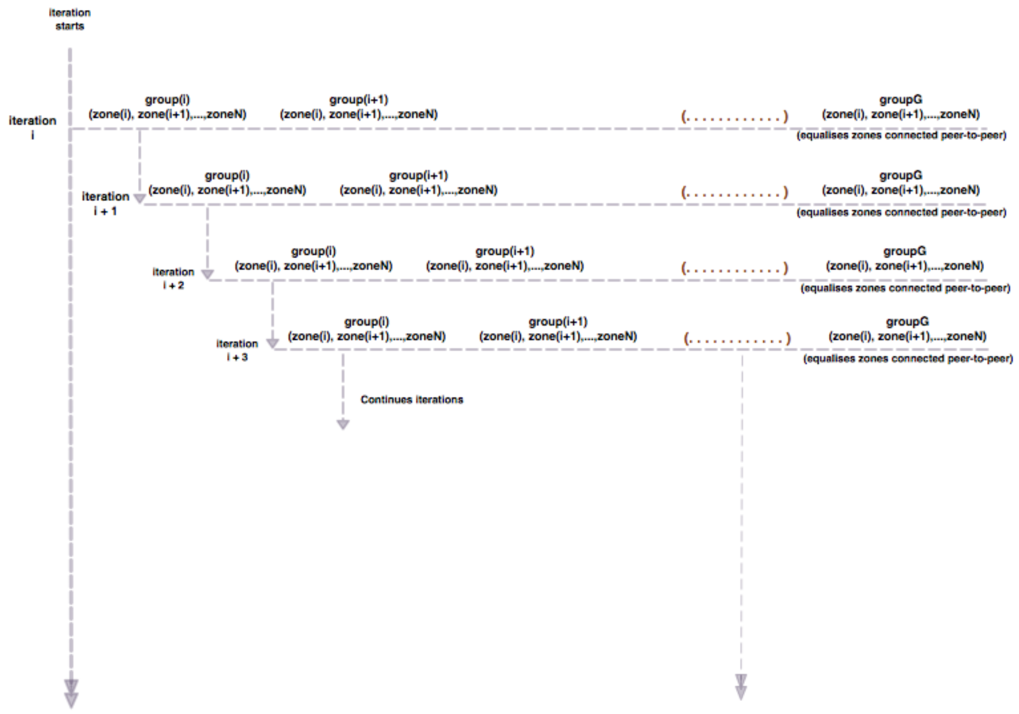

iteration starts
iteration i
group(i) (zone(i), zone(i+1),...,zoneN)
group(i+1) (zone(i), zone(i+1),...,zoneN)
(...........)
groupG (zone(i), zone(i+1),...,zoneN)
(equalises zones connected peer-to-peer)
iteration i + 1
iteration i + 2
iteration i + 3
Continues iterations

Figure 3.7: Flow-Diagram of Peer-to-Peer Communication

Algorithm 7 Graph Partitioning Algorithm

```
1: Suppose, we have $N$ number of zones denoted as $Z$ in each group, where
   $Z = \{Z_1, Z_2, Z_3, \dots, Z_N\}$.
2: Suppose, we have $S$ number of VMs denoted as $VM$ in each zone, where
   $VM = \{VM_1, VM_2, VM_3, \dots, VM_S\}$.
3: Suppose, we have $X$ number of groups denoted as $G$, where
   $G = \{G_1, G_2, G_3, \dots, G_X\}$. Each $G_i$ have a set of two or more zones.
4: Let, $r_i(t)$ denote the current response time at zone $Z_i$.
5: Let $G_i r_i(t)$ denotes the current total average response time at $G_i$.
6: Let $G_i l_{ij} = 1$ if $VM_i$ is in $Z_j$ of $G_i$, otherwise 0. It gives the location of $VM_i$ at
   instant time $t$.
7: At time $t_i$, $VM_i$ collects average response time of current zone $r_i(t)$, according to
   equation (3.2).
8: $VM_i$ calculates total average response time of $G_X Z_N$ derived by $G_i\bar{r}(t) = \frac{\sum_{i=1}^{N} G_i r_i(t)}{N}$
   where $G_X Z_N = \sum_{i=1}^{X} G_i\bar{r}(t)$ and checks if $G_i r_i(t) < G_i\bar{r}(t)$.
9: If true $VM_i$ finds $max(G_X Z_N \bar{r}_i)$ and picks it as candidate zone to move.
10: $VM_i$ will move to candidate zone with the probability $\frac{\bar{G_i r} - r_i}{\bar{G_i r}}$.
11: Repeat from Step 7 to Step 11.
```

Needless to say the last two variants (Alg. 6 and 7) have more number of zones and takes the higher number of iterations (longer time) to converge as a whole. Fig. 3.8 shows all the VMs in clusters are connected to each other in groups. All the directly connected zones in each group are equalising their average response time within the group or with the any of the connected zones in the linked groups at the specific point of time.

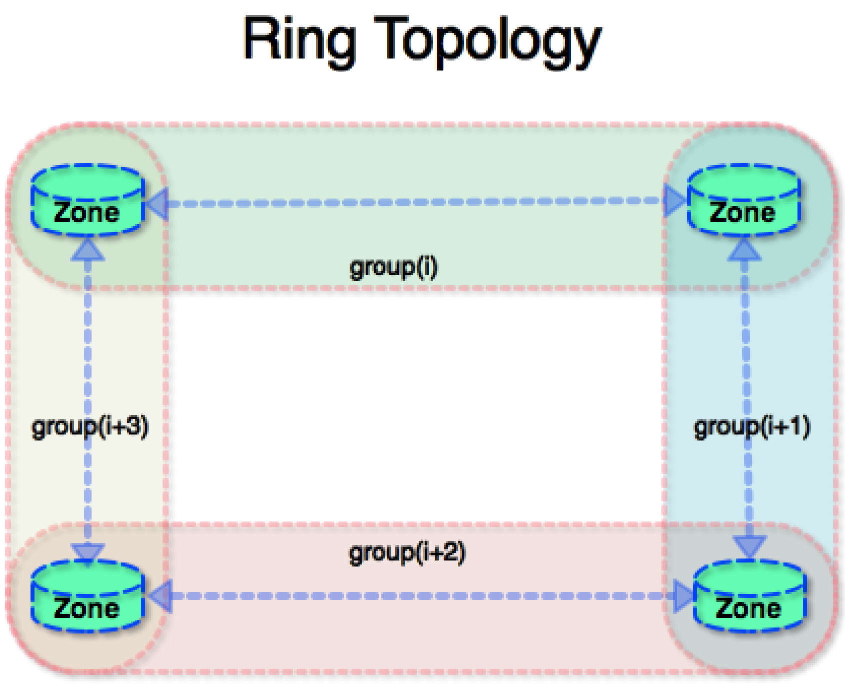


Figure 3.8: Zones Partitioned by Groups

3.3 Architecture Overview

The infrastructure to implement the algorithms consists of multiple instances (In this sections we will address all VMs as instances) in the Google cloud platform. Initially, every zone will be equipped with different instances which are prepared to obtain certain purposes in the complete infrastructure. All the different instances are designed to constantly in sync with each other in a zone and are in a collaborative way serving as a distributed web service. The following figure (Fig. 3.9) gives a brief knowledge of the model of the infrastructure where each instance is independently and autonomously choosing their next zone to migrate to.

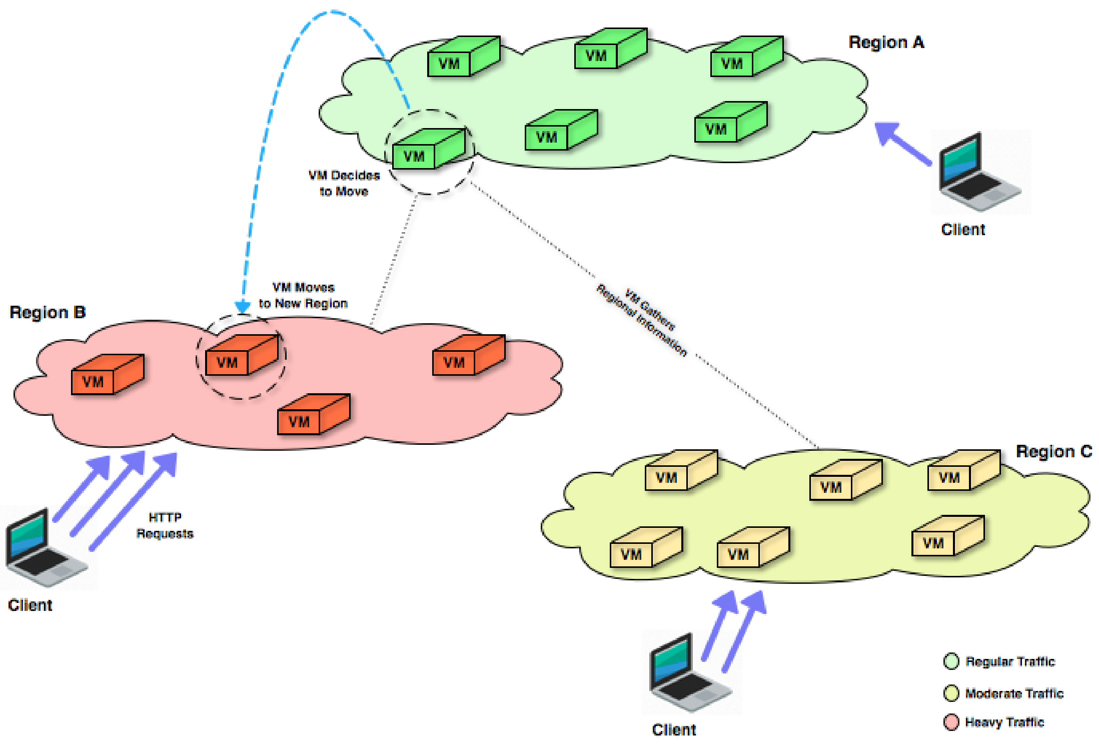


Figure 3.9: Model of the System Infrastructure

3.3.1 Infrastructure requirements

As we discussed earlier in Sec. 2 and 2.5 we are deploying our web service in GCP and Ubuntu images will be optimized to implement as instances' OSs. All the instances are running the latest version of Ubuntu and have been slightly adjusted to give all the system a general initial basis. Multiple packages are installed in each instance and these packages are essential for the systems configuration management. The packages such as **Git**, **unzip**, **curl**, **jq** and **bc** are installed in every instance despite the purpose of the instances in overall infrastructure. Additionally, some other packages are installed such as **Consul**, **Consul-Haproxy**, **HAProxy** will be set up accordingly to the systems' purpose which is discussed in the latter sections.

Latest version of Ubuntu has time synchronization built in and activated by default using **systemd's timesyncd** service. Until recently, most network time synchronization was handled by the Network Time Protocol daemon or **ntpd**. This server connects to a pool of other **NTP** servers that provide it with constant and accurate time updates. Ubuntu's default install now uses **timesyncd** instead of ntpd, which connects to the same time servers and works in roughly the same way, but is more lightweight and more integrated with systemd and the low-level workings of Ubuntu. Servers OSs are by default set to the **UTC** time zone, which is Coordinated Universal Time, the time at zero degrees longitude. Consistently using Universal Time reduces confusion when our infrastructure spans multiple time zones. To avoid having unsynchronised clock issue in Ubuntu the time-zone of each instance is set to **Europe/Oslo** by using the following command:

```
sudo timedatectl set-timezone Europe/Oslo
```

The above command ensures that NTP time is set to all the systems and the time is synchronised on the different geographically located instances. We are also using Ubuntu 16.04 and created an image called **ubuntu-console** for this project. As a machine-type we have used **n1-standard-2** for load-balancer instances and **n1-standard-1** for consul-master and nat-gateway instances. For worker instances we are using lightweight **f1-micro** machine-type.

3.4 Regional Setup

Each region will be described as an individual data centre located in different geographical location in GCP infrastructure. The data centres are referred as DC_i where i = {1,2,3,....m}. Each region considered to have a **Consul-Master-DC**, a **LB-DC**, a **NAT-Gateway-DC** instance and multiple **Worker-DC-X** where X = {1,2,3,...,n} (recommended above 30). Fig. 3.10 shows the setup of all the instances in a particular region.

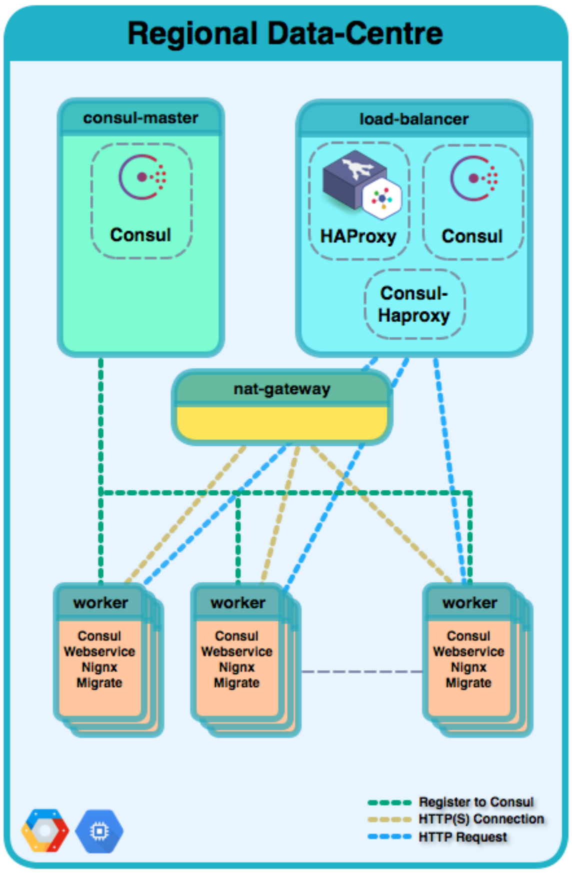


Figure 3.10: Illustration of Regional Data-Centre Setup

3.4.1 Cross-region HTTP load-balancer setup

As discussed earlier (Sec. 2.5.1), the project needs to have a specific IP address that will successfully forwards requests to the instances that is closest (geographically) to the users. In order to do so, we create a HTTP(S) load-balancer that forwards traffic to instance groups. This sort load-balancer works with instance groups and we need to set up groups with lb-dc instances for each region in our infrastructure. To start with the setup we are creating an IPv4 global static external IP addresses for cross-region load-balancer with the following command:

```
gcloud compute addresses create <address_name> --ip-version=IPV4 --global
```

Next we are creating intances groups for each our regions for setup:

```
gcloud compute instance-groups unmanaged create <instance_group> --zone <zone>
```

We have created individual group names for different regions. Next, the lb-dc instances of each region are then added to these instances groups respectively:

```
gcloud compute instance-groups unmanaged add-instances <instance_group> \
--instances <lb-dc_name> --zone <zone>
```

We also create health check options, as GCP health checks determine whether instances are "healthy" and available to do work.

```
gcloud compute health-checks create http http-basic-check
```

For each instance group we define HTTP service and map a port name to the relevant port (in this project we are using port 80):

```
gcloud compute instance-groups unmanaged set-named-ports <instance_group> \
--named-ports http:80 --zone <zone>
```

The web-map back-end services needed to be created and their parameters need to be specified. We are setting protocol field of the back-end service to HTTP (we are using HTTP to go the to web services) and adding http-basic health check to it:

```
gcloud compute backend-services create <web_map> --protocol HTTP \
--health-checks http-basic-check --global"
```

We add the instance groups as back-ends to the back-end service we created. It defines the capacity (maximum CPU utilisation or maximum queries per second) of the instance groups it contains. In this setup, we are setting the balancing mode to be CPU utilisation, the maximum utilization to be 80%, and the capacity scaling to be 1 for each region. Capacity scaling can also set to 0 if we want to drain a back-end service.

```
gcloud compute backend-services add-backend <web_map> \
--balancing-mode UTILIZATION --max-utilization 0.8 \
--capacity-scaler 1 --instance-group <instance_group>-resources  \
--instance-group-zone <zone> --global
```

Now we create a default URL map that directs all incoming requests to all our lb-dc instances followed by creating a target HTTP proxy to route requests to our URL map:

```
gcloud compute url-maps create <url_map> --default-service <web_map>
gcloud compute target-http-proxies create <target_proxy> --url-map <url_map>
```

Finally, global forwarding rules are created to route incoming requests to the proxy using the load-balancer IP address we have created earlier:

```
gcloud compute forwarding-rules create <forward_rule> \
--address <lb_ip_address> --global --target-http-proxy <target_proxy> \
--ports 80
```

The initial cross-region HTTP load-balancing is done when all the lb-dc instances' IP addresses are propagated with the url-map. Appendix: B.1, are created in order to automate the setup of creating and adding instance groups for the initial infrastructure setup.

3.4.2 Google cloud DNS setup

In this project, the Google cloud DNS setup requires to have a common URL to reach the **forwarding-rules** of the cross-region load-balancer. As we discussed in Sec. 2.5.1, we can add multiple potential URLs that are all identified by the same destination address. The request to reach any of the servers will be chosen on the basis of their geographical location, i.e., whichever server is located geologically close to the origin of the requests. This is done by using the url-map we have created in the previous section. This is mostly useful as we need to confirm that all the servers closest to the users will be serving first. At first, we need to create managed-zone (**project-brainiac**) in Google cloud DNS. Then we need to add the A and CNAME records to that managed zone. The following commands need to run to create and set up the initial state of the cloud DNS. Before all, we need to make sure we have obtained a registered domain. For this project the domain name is **project-brainiac.eu**.

```
gcloud dns managed-zones create project-brainiac \
--description "A NSA maters thesis project DNS" \
--dns-name project-brainiac.eu
```

The **dns.sh** script (Appendix: B.2) runs and dynamically finds all the load-balancers. The following snippet is running at the initial setup when the new load-balancer instances are created and their external IPs are added as the **A** record to the domain name of the managed-zones at the end of the script.

```
ALL_OLD_RECORDS=$(gcloud dns record-sets list --zone=$project |
tail -n+2 | awk '{print $4}' | grep -oE "\b([0-9]{1,3}\.){3}[0-9]{1,3}\b" |
while read line; do echo "\"${line}\""; done | tr '\n' ' ')
ALL_NEW_RECORDS=$(gcloud compute forwarding-rules list |
grep http-cr-rule | awk '{print $2}')
eval "gcloud dns record-sets list --zone=$project"
eval "gcloud dns record-sets transaction start --zone=$project"
eval "gcloud dns record-sets transaction remove --zone=$project
--name="project-brainiac.eu." --type=A --ttl=300 $ALL_OLD_RECORDS"
eval "gcloud dns record-sets transaction add --zone=$project
--name="project-brainiac.eu." --type=A --ttl=300 $ALL_NEW_RECORDS"
eval "gcloud dns record-sets transaction execute --zone=$project"
```

Every time the transaction starts a file named **transaction.yaml** is created. All the actions are added to this file. After executing the transaction changes are then inserted into the records. Since the CNAME will not be changed always for managed-zones, we can add this by default to the project-brainiac zone before executing the transaction yaml. The following step is not mandatory.

```
gcloud dns record-sets transaction add --zone project-brainiac \
--name www.project-brainiac.eu --type CNAME project-brainiac.eu. \
--ttl 300
```

Now we have a set of A records attached to our domain name. Every time the URL **project-brainiac.eu** or **www.project-brainiac.eu** is called in a browser by a user from any geographical location, the request will be forwarded to the closest load-balancer that is nearest to that user request. This makes sure we exhaust the closest zone first before we forward request to the next nearest servers.

3.4.3 Service account setup

A service account is a special account that can be used by services and applications running on Google Compute Engine instance to interact with other Google Cloud Platform APIs. Applications can use service account credentials to authorize themselves to a set of APIs and perform actions within the permissions granted to the service account and virtual machine instance. In simple words, a service account is an identity an instance or an application is running with. We can use service accounts to create instances and other resources. If we create a resource using a service account, that resource is then owned by the service account. Service account of an existing instance can also be changed. An instance can have one service account only.

In order to make our instances more versatile and dynamic, we have to make sure that every instance has access to Google cloud platform and its resources. This will make the instances to observe and run control command independently without any help of centralised systems' help. Once created we need to grant roles for that service account. Service accounts are needed to call with scopes. A service account usually consists the following format (in a form of an email) and can be created in IAM service account in Google console:

```
[SERVICE-ACCOUNT-NAME]@[PROJECT_ID].iam.gserviceaccount.com
```

In this project, a service account is created and granted owner permission so it can grant all the authentication permissions necessary to the instances in order to access the resources. As a scope, we select the Google cloud-platform. We need to ensure that every instance creation by gcloud SDK contains this two options.

```
gcloud -q compute instances create example-instance --image <image_name> \
--machine-type <machine_type> --zone <zone> \
--service-account [SERVICE-ACCOUNT-NAME]@[PROJECT_ID].iam.gserviceaccount.com \
--scopes cloud-platform
```

3.4.4 Consul-Master instance setup

To start with each **consul-master-dc** instance will be equipped with **Consul** software as this server will be used to register and de-register worker instances in a particular zone. To set up the consul, it is set up by downloading the zip version of the binary package of consul and copied in a /**usr**/**local**/**bin** directory of the Linux OS. The following snippet is the part of Appendix: C.1 which is runs as a part of the start-up script while creating the consul-master-dc instances in each zone.

```
wget https://releases.hashicorp.com/consul/1.0.6/consul_1.0.6_linux_amd64.zip
unzip consul_1.0.6_linux_amd64.zip
mv consul /usr/local/bin
mkdir -p /var/lib/consul/
mkdir -p /usr/share/consul
mkdir -p /etc/consul.d
```

In addition, several consul data directories will be created for the consul to generate logs and create services. Next, we are setting up consul so that it can run as a service in the background. In order to do that, we add the file **consul.service** to the **/etc/systemd/system/** directory. The content of the folder would be as follows:

```
[Unit]
Description=Consul
Documentation=https://www.consul.io/

[Service]
ExecStart=/usr/local/bin/consul agent -server -ui -bootstrap-expect=2 -data-
    dir=/tmp/consul -node=${consul_hostname} -bind=${consul_ip} -datacenter $
    {consul_dc} -config-dir=/etc/co$
ExecReload=/bin/kill -HUP $MAINPID
LimitNOFILE=65536

[Install]
WantedBy=multi-user.target
```

This above consul.service configuration selects the consul-master instance as the consul-server and with the **-bootstrap-expect** option we are letting the servers know that the agent is running in a server mode. Server nodes are responsible for running the consensus protocol and storing the cluster state. The client nodes are mostly stateless and rely heavily on the server nodes. Before a Consul cluster can begin to service requests, a server node must be elected leader. Thus, the first nodes that are started are generally the server nodes. Bootstrapping is the process of joining these initial server nodes into a cluster. The -bootstrap-expect option informs Consul of the expected number of server nodes and automatically starts the leader election process when that many servers are available. In our case, we are keeping the number of selection process between just two consul servers. To prevent inconsistencies and split-brain situations (that is, clusters where multiple servers consider themselves leader), all servers should either specify the same value for -bootstrap-expect or specify no value at all. Only servers that specify a value will attempt to bootstrap the cluster. Consul document recommends 3 or 5 total servers per data centre. Since we are going to deploy a very small infrastructure, the number of consul servers are set to be only 2.

In the above configuration the **consul_hostname**, **consul_ip** and **consul_dc** will be identified by running the following commands respectively in each consul-master instance:

```
consul_hostname=$(hostname)
consul_ip=$(hostname -I)
consul_dc=$(hostname | sed "s/consul-master-//g")
```

The **-node**, **-bind** and **datacenter** options are used to integrate instance name, IP address and data centre name respectively for each consul-master server.

Furthermore, the options **-ui** sets that the user interface to monitor the consul servers will be deployed in the consul-master instances. In order to set this up, we add the following json to the **ui.json** file in **/etc/consul.d** directory (This directory is for .json files that will hold any extra configuration. This a directory for Consul configuration. Consul loads all configuration files in the configuration directory):

```
{
"addresses": {
"http": "0.0.0.0"
}
}
```

Finally, the consul.service needs to run at system startup and need to be enabled in the system daemon:

```
sudo systemctl daemon-reload
sudo systemctl start consul.service
sudo systemctl enable consul.service
```

3.4.5 Load-balancer instance setup

Similar to the consul-master-dc server in a data centre, we setup **lb-dc** instances for each zone. We start with setup consul in the load-balancer which is almost identical as consul-master except the **-join** option will be included in the consul.service configuration. This -join option will let the load-balancer know to join the consul cluster by pointing it to the first server (consul-master). The following snippet of the script Appendix: C.2 will run at the start-up of first time the **lb-dc** instance boots up. This snippet is part of the initial setup for load-balancer:

```
lb_hostname=$(hostname)
lb_ip=$(hostname -I)
lb_dc=$(hostname | sed "s/lb-//g")
consul_ip=$(gcloud compute instances list | tail -n+2 | \
grep consul-master-${lb_dc} | awk '{print $4}')
```

And the consul.service file in the /etc/systemd/system/ directory will be created (by running the lb.sh script) with the following content:

```
[Unit]
Description=Consul
Documentation=https://www.consul.io/

[Service]
ExecStart=/usr/local/bin/consul agent -server -join=${consul_ip} -data-dir=/
    tmp/consul -node=${lb_hostname} -bind=${lb_ip} -datacenter ${lb_dc} -
    config-dir=/etc/consul.d/
ExecReload=/bin/kill -HUP $MAINPID
LimitNOFILE=65536

[Install]
WantedBy=multi-user.target
```

It is noticeable from above configuration of consul.service that the **consul_ip** information from the same data centre are extracted by using the Google cloud SDK (gcloud SDK) commands. This is required since we are trying to setup the load-balancer which can easily recognise a consul-master from the same data centre. The **lb_dc** are selected by extracting the load-balancer instance's hostname.

After, enabling the consul.service in the system daemon of the instance it the consul service is restarted. The consul.service of the lb-dc will then join the consul-master cluster and a leader selection process will occur (since the bootstrapping process was set to 2 servers) and it will join as a member/agent of consul cluster for that zone. Next, we need to install HAProxy to the load-balancer instance as it is the requirement to balance the incoming request to the back-end servers. This is done by simple step at the beginning of the startup script **lb.sh**.

The load-balancer can only balance the loads between servers only when it is manually added to its configuration. Since we are working on a dynamic environment where the worker instances will be migrating to different zones independently we need to consider using the service that will discover the services that the worker instances are providing and publish this to the load-balancer. Hence the Consul service is introduced as a service discovery agent. In the load-balancer, we need to setup consul servers to listen for any client hosts that is providing a web services. Upon discovering the agents with services Consul will automatically register that service and publish this to the load-balancer (HAProxy). The load-balancer will then add that host as back-end server dynamically and make it the part of its entire distributed web service. The agent (migrating VM) also need to de-register from the consul-master of the zone that it is leaving from. This requires a inclusive configuration management which starts with setting up **consul-haproxy** to the **lb-dc** instances in each zone. The consul-haproxy is a daemon for dynamically configuring HAProxy using data from Consul. The daemon watches any number of back-ends for updates, and when appropriate renders a configuration template for HAProxy and then invokes a reload command, which can gracefully reload HAProxy. This allows for a zero-downtime reload of HAProxy which is populated by Consul.

Similar to the consul installation we download the binary and copy it to the **/usr/local/bin** directory. This allows the consul-haproxy to run from anywhere in the OS. We create a **consul_ha.cfg** file in the **/etc/consul.d/** directory which contains a template for haproxy configurations:

```
........
frontend http_front
bind *:80
stats uri /haproxy?stats
default_backend http_back

backend http_back
balance roundrobin
{{range .c}}
{{.}}{{end}}
........
```

The above is some of the important snippets of the consul_ha.cfg template file. The http-back block will be populated as soon as consul server register a new agent and that new agent is providing a web service. In this project, we are naming our service as **webserver** which will be discussed in the Sec. 3.4.6.

Finally, the consul-haproxy will be run in order to change the HAProxy configuration every time a new webserver agent is registered.

```
sudo consul-haproxy -addr=localhost:8500 -in /etc/consul.d/consul_ha.cfg \
-backend "c=webserver@${lb_dc}:80" -out /etc/haproxy/haproxy.cfg \
-reload "/etc/init.d/haproxy restart"
```

The above command runs and looks for any webserver agent that is providing in the localhost:8500 of the consul service. The address 127.0.0.1:8500 is used by consul as default to query services. These are address and port of the Consul HTTP agent. The value can be an IP address or DNS address but must include the port. This also is specified CONSUL_HTTP_ADDR environment variable. If an agent is registered it dynamically populate the template from /etc/consul.d/consul_ha.cfg file with all the agent with service named webserver as consul member and add it as back-end server to the haproxy. This webserver name query is specified as the **-backend** option with **"c=webserver@lb-dc:80"** which literally telling the consul-haproxy to add all the agents which are providing the service webserver to the load-balancers port 80. The populated template is then used to replace the old **haproxy.cfg** file which stores the HAProxy configuration and reloads the HAProxy service. This makes the back-end servers added to the configuration automatically. Like we discussed earlier, the servers will also be removed from the back-end configuration once they de-register themselves from the consul cluster gracefully.

This process needs to always run in the background of the load-balancer instances. The command is set to run as background with the following command:

```
nohup <consul-haproxy command> &>/dev/null & disown
```

3.4.6 Worker instance setup

The **worker-dc** setup almost similar to consul servers except it is designed to provide web service to the architecture. Similar to other instances, we start with setting up consul in the worker instances. Instead of running consul as systemd service we run the consul register and join the consul cluster in the same data centre in the background. The process is almost same as running it as consul.service in consul-master-dc and lb-dc instances but simpler to implement. As the worker instances are designed to migrate on demand or certain criteria it is easier to start the process consul cluster joining process in a simpler manner. In order to achieve this, after consul setup we run the command in the worker-dc instances in the background:

```
worker_hostname=$(hostname)
worker_ip=$(hostname -I)
worker_dc=$(hostname | cut -c8-10)

nohup sudo consul agent -server=false -data-dir=/tmp/consul \
-node=${worker_hostname} -bind=${worker_ip} -enable-script-checks=true \
-datacenter ${worker_dc} -config-dir=/etc/consul.d &>/dev/null & sleep 3 \
&& consul join consul-master-${worker_dc}
```

The above command starts by running the consul as a client by setting the option **-server=false**. When the consul command run properly it will then send a request to join the cluster to the consul-master of the own data centre. A consul client can request to join any of the consul servers that are doing the service discovery. When the worker instance joins successfully it will be registered as consul agent of the same data centre. The worker instance also provides service as a back-end server of the distributed web service. We setup **Nginx** as the webserver in these instances.

```
sudo apt-get install -y git unzip nginx
sudo ufw allow 'Nginx HTTP'
sudo systemctl start nginx
```

We write a service definition configuration file in /etc/consul.d/ directory and name it **webserver.json**. We have a service named "webserver" running on port 80. Additionally, we'll give it a tag we can use as an additional way to query the service. The webserver.json file contains the following:

```
{
  "service": {
    "name": "webserver",
    "tags": ["HTTP"],
    "port": 80
  }
}
```

Although the setup of a worker is simpler, there is additional configuration necessary for setting up worker-dc instances. As the limitation of GCP quotas, the ephemeral external IPs are limited to 8 IPs per zone. Ephemeral external IP addresses are available for instances and forwarding rules. Ephemeral external IP addresses remain attached to an instance only until the instance is stopped and restarted or the instance is terminated. If an instance is stopped, any ephemeral external IP addresses assigned to the instance are released back into the general Compute Engine pool and become available for use by other projects. When a stopped instance is started again, a new ephemeral external IP address is assigned to the instance.

As we mostly going to use the private IPs on instances (except lb-dc), it is not mandatory to add an external IP while creating worker-dc instances. It can be avoided by adding the option **–no-address** to the gcloud SDK command. But this means the default NAT IP will not forward any external HTTP/s packets towards internal addresses. We need to ensure that all the worker instances have access to outside the internal network and can access the internet, i.e., we have to enable HTTP/s access to the instances. This enables the worker instances to reach the port 80 and 443 and can reach the load-balancer's IP externally to read the network conditions for migration purpose. A nat-gateway instance will be dedicated to each zone for worker instance to forward its HTTP request and response back. In the next section (Sec. 3.4.7), the detail description of nat-gateway-dc setup has been discussed.

Furthermore, we get a copy of the CSV generator script (csv-gen.sh) and corresponding migration script (migration-uniform-naive.sh/ migration-uniform-informed.sh/ migration-biased.sh/ migration-single-point.sh) in the worker instance and set it to run as a background process every certain interval. Appendix: D.1 runs every 5 secs to gather and calculate average response time in each zone and Appendix: D.2-D.5 script will run between every 5-10 mins depending on which variant of our designed algorithm will be implemented in our infrastructure. Each of the migration scripts are designed to implement our designed variants of algorithm (Alg. 2-5) in our cloud setup in order to observe the performance of the algorithms in real-life test-beds. Thorough clarification of these migration rules implementation in the worker instances will be discussed in Chapter. 4. The migration scripts are running as background process of worker instances:

```
nohup sudo /bin/bash -c 'while [ true ]; do sleep 30; /tmp/project-brainiac/
    sidekicks/csv-gen.sh; done' &>/dev/null & disown
nohup sudo /bin/bash -c 'while [ true ]; do timer=$(( ( RANDOM % 300 ) + 300
    )); sleep $timer; /tmp/project-brainiac/startups/worker-[variant]/
    migrate-[variant].sh; done' &>/dev/null & disown
```

The above background process runs the csv-gen.sh (Appendix: D.1) script in each 30 secs interval and the migration script (depending on the migration scheme we like to implement in the network) in randomly between 5-10 mins.

3.4.7 NAT-Gateway instance setup

In the **nat-gateway-dc** instance creating some prior setup need to be done. This particular instance is designed to forward http packets to and from internal private network. First we have to make sure the firewall-rules for **internal** and **ssh** already exists in the **default** GCP network (we can create a complete set of legacy network using different name [32]):

```
# gcloud compute firewall-rules list
...
default-allow-internal  default  INGRESS   65534    tcp:0-65535,udp:0-65535,icmp
default-allow-ssh       default  INGRESS   65534    tcp:22
...
```

The settings are enabled while creating these instances with gcloud SDK:

```
gcloud compute instances create nat-gateway-dc --network default \
--can-ip-forward --image <image_name> --machine-type <machine_type> \
--zone <zone> --tags nat --metadata-from-file startup-script=./nat-gateway.sh

gcloud compute routes create no-ip-internet-route-dc --network default \
--destination-range 0.0.0.0/0 --next-hop-instance nat-gateway-dc \
--next-hop-instance-zone <zone> --tags no-ip-<dc> --priority 800
```

The above commands create a nat-gateway-dc instance to a specified zone which has the ability to forward IP for that particular. This instance act as a NAT gateway on the default network. The second command creates a route to send traffic destined to the Internet through gateway instance. Setting the priority of this route ensures that this route takes precedence if there are any other conflicting routes. 1000 is the default priority and a value lower than 1000 takes precedent.

While creating a new worker instance in that particular zone without an external IP we only need to add the tag with option **–tags** set to **no-ip-dc** in this way:

```
gcloud compute instances create worker-dc-example --network default \
--no-address --image <image_name> --machine-type <machine_type> \
--zone <zone> --tags no-ip-<dc> --metadata-from-file startup-script=./worker.sh
```

All the worker instances can now reach the internet via the nat-gateway-dc instances. The gateway instance needs an additional configuration in order to forward all the requests/responses. We run a script (Appendix: C.3) while creating nat-gateway-dc instance which will enable the ip4 forwarding by **ip_forward** file in **/proc/sys/net/ipv4/** directory to "1" and adding the POSTROUTING rule of iptables NAT rule to **MASQUERADE**.

```
sudo sh -c "echo 1 > /proc/sys/net/ipv4/ip_forward"
sudo iptables -t nat -A POSTROUTING -o ens4 -j MASQUERADE
```

Now all the instances that have no-ip-dc(*) tags (with corresponding data-centre) can access the internet via the nat-gateway-dc(*) instance of the same region.

3.4.8 Migration procedure

To implement the algorithms, we are creating different migrations scripts in this part of the project, which are designed to initiate worker instance' behaviour according to corresponding variants of algorithms at the system startup. This ensures that the instances will apply algorithms uniform-site (naive/informed) migration, biased migration, single-point migration depending on which behaviour we like to promote in each instance. To start with the migration procedure in the machines, a script **csv-gen.sh** (Appendix: D.1) will run at intervals to gather average response times from each active load-balancer instances in the network. The information of individual back-end servers of load-balancer which are providing the web services are gathered by reading through the HAProxy socket using the following command:

```
curl -sSL 'http://$lb_ip/haproxy?stats;csv;norefresh' | cut -d "," -f 2,61 | \
column -s, -t | grep worker | awk '{print \$2}'
```

This particular command gets the statistics of HAProxy in a CSV format and only collects back-end webserver name and its response time at a particular moment. The average response time of each data-centre is then calculated by using *sum of all the back-end servers' response time/number of back-end servers*. The script will also create a log file and keep running in the background for future migration decision purpose. The different migration scripts to invoke migration behaviour in the worker instances are described in the following sections.

Uniform-site (naive) migration procedure

Appendix: D.2 runs on worker instances which implies the Alg. 2 in the project architecture. The script initiates such behaviour in the instances where it migrates to a candidate zone which is selected randomly. This means a worker instance will run a check if its own data-centre's average response time is lower than the response time threshold (i.e., *sum of all the zones' average response time/number of zones*). If it does, then the worker instance will decide to migrate and run a selection method where it randomly chooses one of the other zones as candidate region to move. Next, the worker creates a new worker instance in that candidate zone and gracefully removes itself from the consul-master of its own data-centre. This part is done by running a simple command in the worker instance:

```
consul leave -http-addr=127.0.0.1:8500
```

This above command runs to gracefully remove the worker from the consul-cluster by sending a notification to the port 8500. Without this particular command, the consul-server will not recognise that the worker instance has been detaching itself and HAProxy will continue to forward HTTP requests to this instance as consul-master will never update the back-end server information. The graceful detachment process makes the consul-master remove the instances from its cluster and let the load-balancer to restart its settings with the current alive back-end servers.

At the final step of this migration procedure, the worker instance will shut-down and delete itself from the infrastructure. The new instance in the candidate zone runs the same setup process at the bootup and behave similarly in an autonomous manner as its originate instance.

Uniform-site (informed) migration procedure

Similar to previous procedure uniform-site informed migration also applies the same sort of behaviour to the worker instances. Appendix: D.3 applies the designed algorithm (Alg: 3) to the worker instances in this scenario and create the environment where worker instance will choose their candidate region based on which region/s have the average response time higher than the worker's current region. The procedure is similar to the uniform-site naive migration scheme but in this case, the workers are given an option to inspect and calculate the network situation more effectively. In this migration procedure, worker instances are more aware of network condition of other data-centres. This makes the worker take a better decision when it comes to choosing a candidate region. The information aspect of this behaviour makes the infrastructure's average response time saturate in a better manner. Instead of migrating naively or randomly to any data-centre workers are designed to make smarter choices in this case. Similar to naive migration, the worker instance creates a worker instance to the candidate region. If there are more than one potential candidates, worker chooses the candidate randomly. Finally, the worker removes itself from its own data-centre's consul cluster gracefully and delete itself from the infrastructure.

Biased migration procedure

In this migration scheme, the worker instances can take further steps to implement the biased migration algorithm (Alg. 4). In addition, to gather and calculate average response time of all the data-centres on different regions, worker instances are also capable of calculating the probability to choose a candidate region. The probability is calculated based on the corresponding regions' average response time and this probability is then used to compare the performance of each region separately. Several individuals steps have been taken to measure this probability for each data-centre. The candidate region chosen by the worker is the data-centre that has the closest probability to migrate to. This means the probability that is closest to moving (probability = 1). The probability margins to select a candidate is between different data-centres are very little and this makes the biased migration procedure more sophisticated migration scheme for a system design. Appendix: D.4 applies this migration behaviour to the worker nodes.

Single-point migration procedure

The single-point procedure is simple but also effective schema that implies the candidate region selection more realistic way. The procedure involves a selection process where the worker instance calculates the average response time of the data-centres and choose the candidate region to migrate based which region has the maximum average response time. This is a real-life test-bed implementation of Alg. 5 and is added as a behaviour to the worker machines with the **migrate-single-point.sh** script (Appendix: D.5). The candidate zone now then gets a new instance if the worker instance decides to migrate (considering its own data-centre's average response time under the recommended threshold). Once decides, migration procedure takes place and worker instance remove itself from own data-centre.

3.4.9 Initial setup script

Finally, a script is provided to initially run all the above scripts in steps to setup the Google cloud for a real-life test-bed scenario. The automate (Appendix: E.1) script takes the user inputs as variable to initiate an automated process to start up a distributed web service with autonomous VM (and implement the corresponding migration script) in the Google cloud.

In the next chapter, we are conducting multiple tests based on simulation to analyse the performance of our designed algorithms. We also administer thorough experiments in real-life test-bed scenarios and evaluate the performance of the proposed algorithms.

Part III

Conclusion

Chapter 4

Results and Analysis

In this chapter, we are discussing the simulation and real-life test-bed results of the designed algorithms. First, we implement them in the simulation using python to observe the performance of algorithms in various distinctive situations. The graphs are created for the simulation to evaluate the performance maps in order to distinguish the best possible outcome of the designed variant of the general evolutionary algorithm. Then the algorithms are implemented in real-life test-bed using GCP and gather the numerical data to asses the behaviour of the autonomous VMs in with the evolutionary game theory. The results are then compared to each other and analysed.

4.1 Results: Test-Simulation

Using Erlang-C formula with evolutionary game theory, several test simulations are conducted. Some initial parameters are being set accordingly for all the variants:

```
The initial number of VMs for each zone = 10
The Initial service rate for each zone = 15
```

Table 4.1: Request Rate ID Sets in Periodic interval for Test Simulation

Request Rate	Algorithms	Z1	Z2	Z3	Z4	Z5	Z6
Set A	Uniform-Site Migration, Biased Migration, Single-Point Migration	70	35	60			
Set B		35	70	60			
Set C		70	60	35			
Set A	Peer-to-Peer Communication	70	30	35	40	50	60
Set B		30	35	40	50	70	60
Set C		35	40	50	60	70	30
Set A	Graph Partition	70	30	35	60		
Set B		30	35	70	60		
Set C		35	70	60	30		

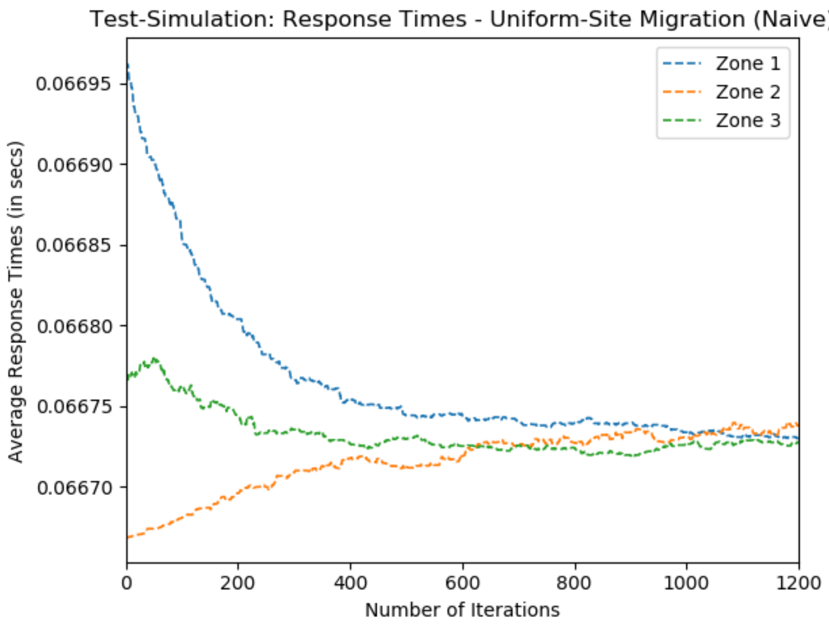
We conducted the test-simulations for Uniform-site (naive/informed), Biased and Single-point migration with 3 zones only. For simulations with Peer-to-Peer communication and Graph partition are conducted with 6 and 4 zones respectively. Table. (Tab. 4.1) shows the set of request rates for different zones in individual simulations. In the table zones are marked as **Z** and request rate sets are indicated as **Set**. The static request rate sets refer to the scenario when the incoming HTTP requests to a zone remain static in entire process where dynamic request rate sets are referring to the situations when the rates of requests are changing after a fixed amount of time (in this test in iterations). Based on the number of zones for each simulation a unique set of iteration time has been set for different scenarios. Each iteration is considered to be 10 times the number shown as X-axis of all the graphs for a better perspective. Average response time will be referred to as **Avg. Resp. Time** for rest of this chapter. Graphs will be presented to latter sections to visualise the performance of the algorithms in the controlled simulation environment, where X and Y axis's of each plot refers to the number of iterations of the test and avg. resp. time (in secs) respectively.

4.1.1 Results: Simulation of uniform-site migration (naive)

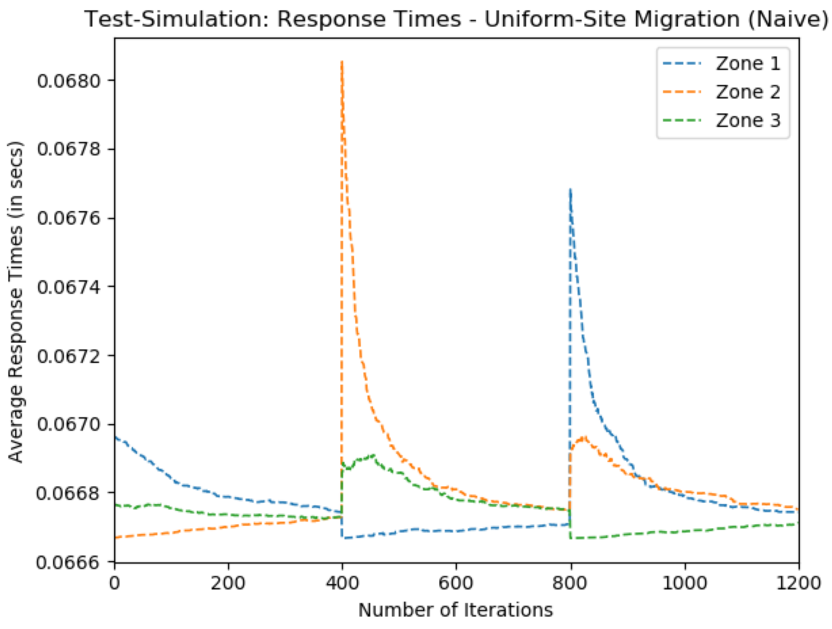
In this test-simulation, the plots are conducted by calculating the number of VMs in per zone and the request rate incoming to the zones the VM is located to. On each iteration, a VM can be migrated to a new zone if certain conditions are met as we discussed in Sec. 3.2.1. The naive Uniform-site migration scheme (Alg. 2) has been implemented in this simulation. Fig. 4.1a and 4.1b represents the test-simulation results in graphs with static and dynamic request rates respectively. The performance of this algorithm is observed in a better way with these graphs. As we described before in Sec. 3.2.2, in this particular algorithm, the migrating VM does not consider the network condition while choosing a candidate zone to move. The candidate zone is rather selected randomly by a VM instance and the avg. resp. time of other zones are not taken into account. This migration process is considered to be headless or aimless. The random migration pattern affects the convergence time of the entire network which we can observe in Fig. 4.1a. The repeated action of random migration does make the system converge after a certain number of iterations due to the nature of moving of the VM to different zones. The convergence period is much longer with this variant of the algorithm as the VMs' migration behaviour does not follow any specific rule to reach a goal. The system does not work as a whole yet the convergence takes place since the VMs in the most affected zone will not consider moving to a different zone. At one specific moment, this results in the most affected zone in the network will gain an extra VM and continue to keep it in the zone until its avg. resp. times get lower than the threshold. The newly migrated VM (or the older VMs) at affected zone have lower chance to migrate to another location (zone) as it will calculate its own zone to be the most affected zone in the migration selection process. After a number of iterations, the VMs will be redistributed among all the zones and the avg. resp. times at all zones becomes stable. This result shows emergence behaviour in the system as we can observe the graph showing in Fig. 4.1a are starting to converge after almost 9000 iterations (as each 100 represents 1000 iterations in the graphs). A sample of calculations of response times in all zones after 8800 iterations are shown below:

```
VMs in zone_1 : 11
VMs in zone_2 : 8
VMs in zone_3 : 11
resp_time_zone_1 : 0.06676504898677925
resp_time_zone_2 : 0.06670175909994472
resp_time_zone_3 : 0.06669546394754529
Iteration: 8800 / 12000
```

The system becomes stable and stays in the same state after the 10000 iterations until the end of the test. Despite the random naive movements of the migrating VMs, the system manages to converge in a stable network condition as a whole. The stabilising procedure tends to be distributed among all 3 zones in the test-simulation network. In the graph it is observed that all the zones are slowly stabilised in the same way, i.e., keeps the similar pattern, especially in zones 1 and 2. Although these two zones have the highest and lowest request rates respectively and migration are randomly placed, after a few iterations, the zones started to show emerging pattern. Zone 3, on the other hand, has the moderate request rate and the avg. resp. time is neither increases nor decreases for this zone. Running the same simulation multiple times might result in a different pattern in the graph.



(a) Static



(b) Dynamic

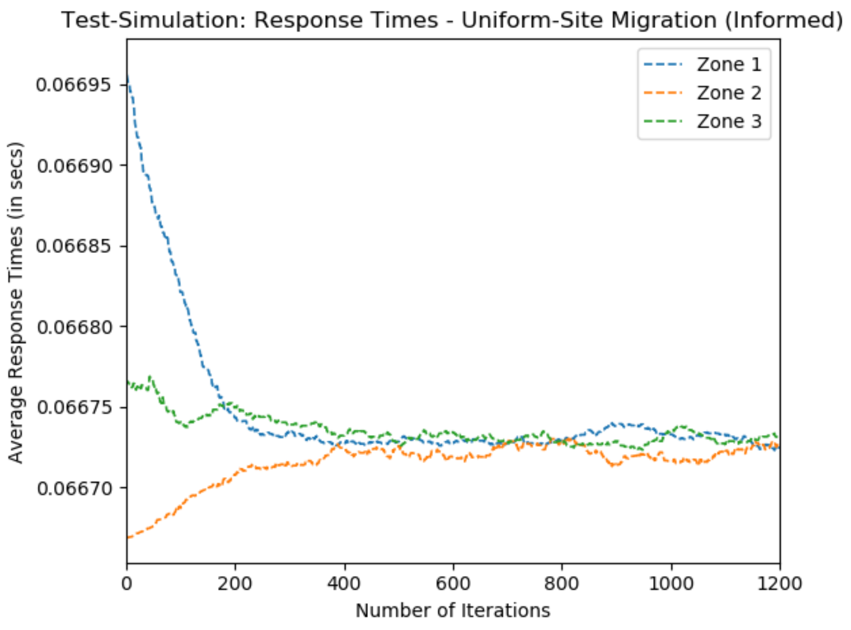
Figure 4.1: Simulation of Uniform-Site Migration (Naive)

Conducting the similar experiment with dynamically changing request rates in the simulation results the graph in Fig. 4.1b. With dynamic request rates, the simulation results starting to converge similarly to the static simulation. It is noticeable that the system converges at almost 4000 iterations in Fig. 4.1b. When the request rate changes for each zone the avg. resp. time changes and immediately starts to show the emergence behaviour in the system. With the same amount of iterations, the graph shows the same pattern as first part of the request rate set (set A). The total avg. resp. time of all the zones comes to the same level and continue to do so until the third set of request rate is introduced. Similar to previous sets, the proposed evolution algorithm starting to show the similar behaviour for the infrastructure. A significant difference in this test than other variant test is that the stabilisation of the whole network needs a higher number of iterations for each set (almost never converges completely with 4000 iterations). This is due to random movement of the VMs in the network which cannot stabilise a single affected zone quickly enough.

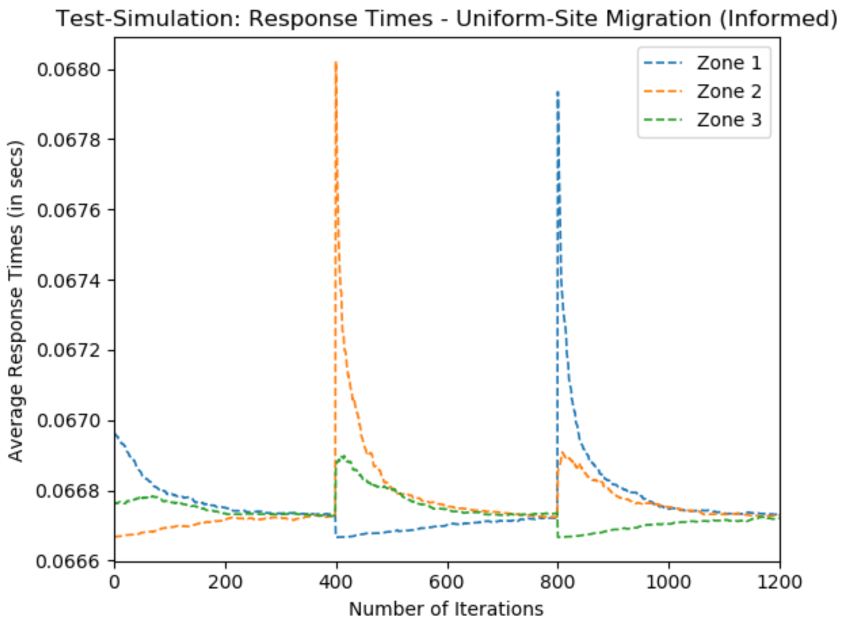
4.1.2 Results: Simulation of uniform-site migration (informed)

In this test simulation, the VMs are given a better migration property than naive uniform-site migration. According to this variant of algorithm specification, the VMs are moving to the zones which have only the worse avg. resp. time than their own data-centre. The characteristic of VMs gives a better strategy to implement the evolutionary game theory in a test-simulation. The graph in Fig. 4.2a represents the static the performance of Alg. 3 in the simulation with static request rates in all the zones. It is observed in the static graph with the informed uniform-site migration, the total convergence of the system is faster than the previous test (Sec. 4.1.1). This is considered to be a better approach to pick a candidate data-centre in the network, as the migration steps of the VM has strategic option to find the performance of its own zone's performance against other connected zones. Having a choice to find the more affected zone/s, migrating VM can make a better independent decision. This impact the network condition of all the zones in the infrastructure, especially the more affected zones after a period of time. In each step, VM can pick one zone that has only worse avg. resp. time. In case of multiple potential candidate zones, VM can pick one randomly. Some randomness are still included in this test. Since the VMs are given a better behaviour property, the convergence steps are shorter in this simulation as we can notice in Fig. 4.2a. Z1 and Z3 start to receive VMs from Z2 since it has the lowest avg. resp. time. In result, both more affected zones (Z1 and Z3) starts to show better response times in the graph. Due to randomness, both affected zones can receive the similar amount of migrating VMs or it can be only one zone gets all the VMs. Running the same simulation as Fig. 4.2a, the system shows the almost identical results. The system gets stable (almost with same response times) at about 4000 iterations, which is a bit faster (ca. 1000 iterations) than the previous test. The system shows stability throughout the rest of the simulation.

Fig. 4.2b represents the test including dynamic request rate in all the zones. We implemented the same principle of choosing the worse candidate zone (randomly) with dynamically changing sets of request rate. The changes of request rates do have much impact on the convergence of the system as we can notice it converges almost after the same iterations in the second and the third sets (set B and C) of request rates. As usual for the first set of requests, the avg. resp. times are stable at 4000 iterations, just before the second sets have been implemented. As it is shown in graph (Fig. 4.2b), the zone with the new highest rate (Z2) becomes unstable and have a rise in response time. It is observable, that on the first set (0-4000 iterations) the response time of the zone with the moderate rate (Z3) was stable until the second set started. Here the zone with the lowest rate (Z1) dramatically dropped and started to increase the response time in a steady manner. The curve for this zone starts to slowly emerge to the convergence point, where the other two zones started to gets stabilised from almost 4200 iterations. In the rest of this set two most affecting zones gives the similar curves which indicate that they are stabilising equally. The stabilisation in the second set happened quicker than the first set. At the third set of this simulation, the rates are changed again and the network shows the same emergence behaviour. It is considered, that repeating this behaviour multiple times will not affect the outcome of the test.



(a) Static



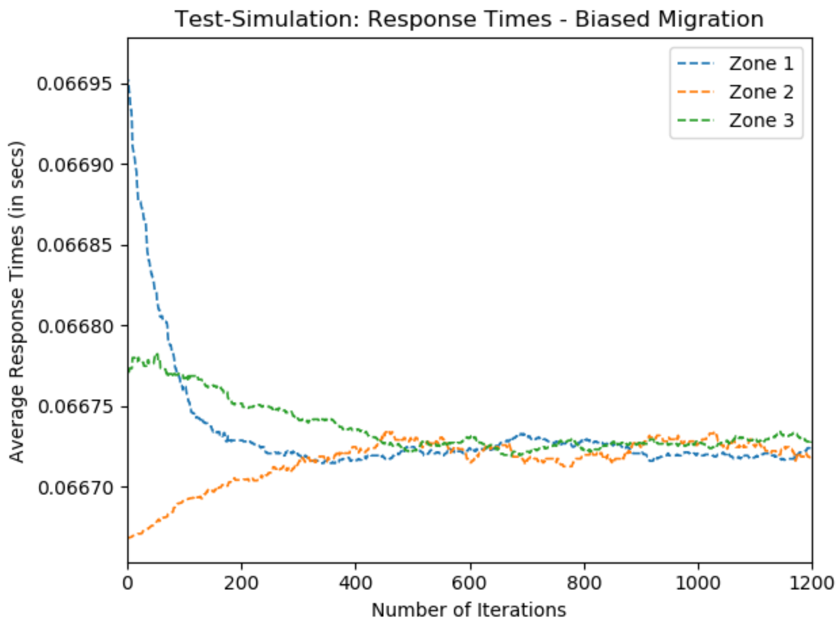
(b) Dynamic

Figure 4.2: Simulation of Uniform-Site Migration (Informed)

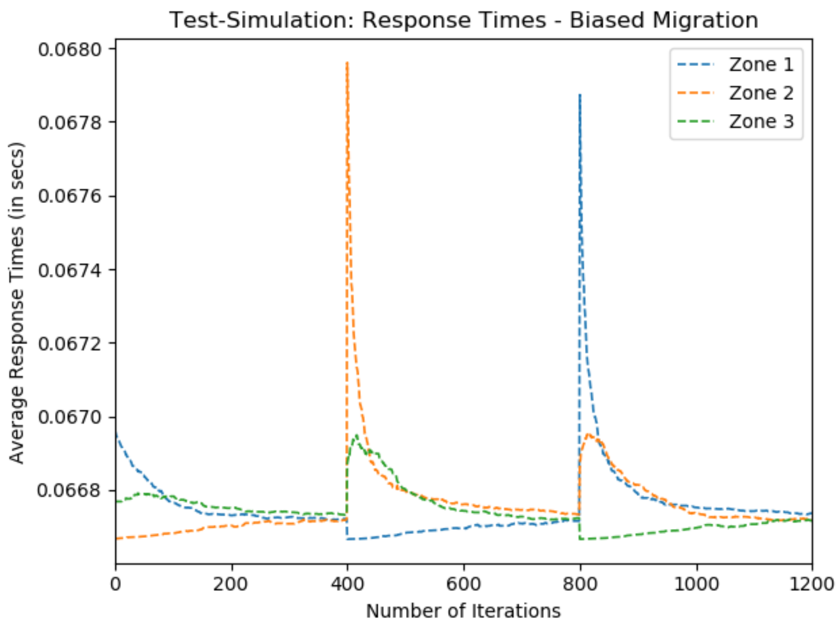
4.1.3 Results: Simulation of biased migration

In this part of the simulation, the VMs have been given a smarter approach to deduct the network condition as a whole instead of making a decision to migrate based on each affected zones' conditions. The biased migration scheme is distinguishable than other variants of the evolutionary algorithms as it chooses the candidate zone with some probability. Hence the name Biased is a good fit for this algorithm. In the simulation, when the incoming request rates are infused, the VMs checks for the highest probability to move to each affected zone with a simple calculation. The effect of VM migration depending on the smallest probability different between the zones are observed in the static biased migration graph (Fig. 4.3a). As we can notice that the Z3 has the lowest incoming HTTP request rate of all three zones and Z1 has the highest amount of request rate during the entire test-simulation period. Z1 starts to stabilise slowly as most of the VMs are migrating from Z2. Surely, VMs in Z2 decides to migrate more often than of Z3. When it does it checks the highest probability to be chosen as candidate zone. In addition, it also picks a threshold for probability for all the network together and picks the zone which is closest to that. This ensures a slow and steady step towards the stability of the response times of all the zones. This simulation is a better test of the general evolution algorithm, as it does not picks the candidate by its current condition but the condition of all the individual conjoint zones, i.e., the entire framework. The approach, however, gives similar performance in the test as it shows in Fig. 4.3a except the curve representing Z1 and Z3 seemingly have more interplay between them, than of Z2 curve. This means, in this test, the most affecting zones get stabilisation in response times among them first before it balances with the zone with the lowest traffic (in this simulation Z2 with lowest request rate). However, Z2 stabilises the response time slowly to the convergence point and at almost 4000 iterations all the curves merge into one point. The response times in all zones holds to that position until the simulation has been completed. There are some fluctuations in response times noticeable around 7500-8500 iterations, after which the system stabilises again.

Fig. 4.3b were conducted using dynamic request rate sets. In the first set of request rate, the system shows the equivalent behaviour as the static rate. The zones with web-services start with three different avg. resp. times depending on the number of requests per zone. Similar to static test-simulation, the two most affected zones (Z2 and Z3) becomes steady in the same fashion. The curves between these two zones overlap at several points and slowly merge with Z1 at the convergence point. The system remains stable and an equilibrium is merged in the network. At almost 4000 iterations, all the nodes are met in the same point (at almost 0.0668ms) and continues until the second set of request rates are introduced. At this point, Z2 becomes the zone with highest avg. resp. time and starting to stabilise with Z3 first. These two zones continue to show the similar curves until it matches the Z1's response time. The pattern repeats itself with the third set (set C) of rates as well. The convergence of this simulation has a smoother curve in the graph, especially between the affected zones as both get migrated VMs from zones with least response time. This represents the implementation of probability to choose the candidate zone that has a better effect on response time equalisation process. Increasing the number of zones in the network (5 or more zones) can give us a better view in this matter.



(a) Static



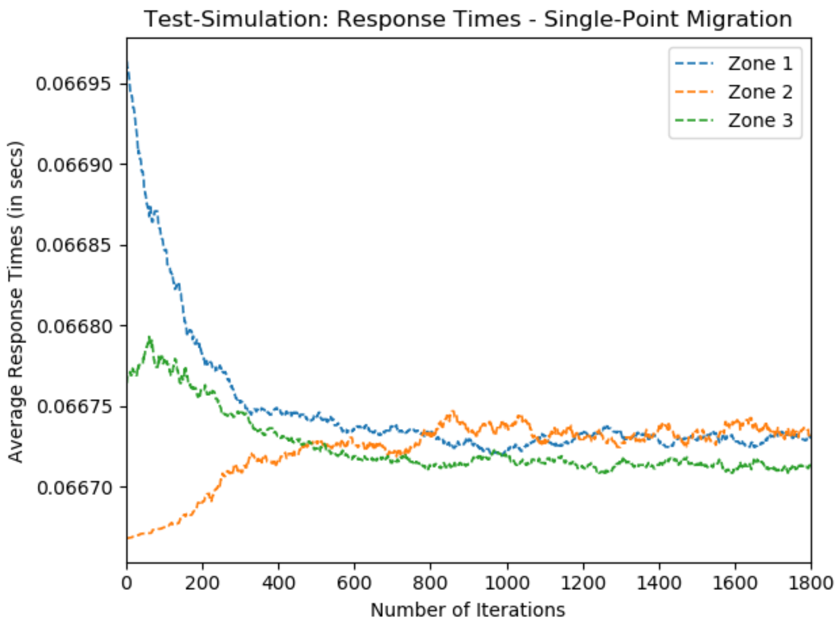
(b) Dynamic

Figure 4.3: Simulation of Biased Migration

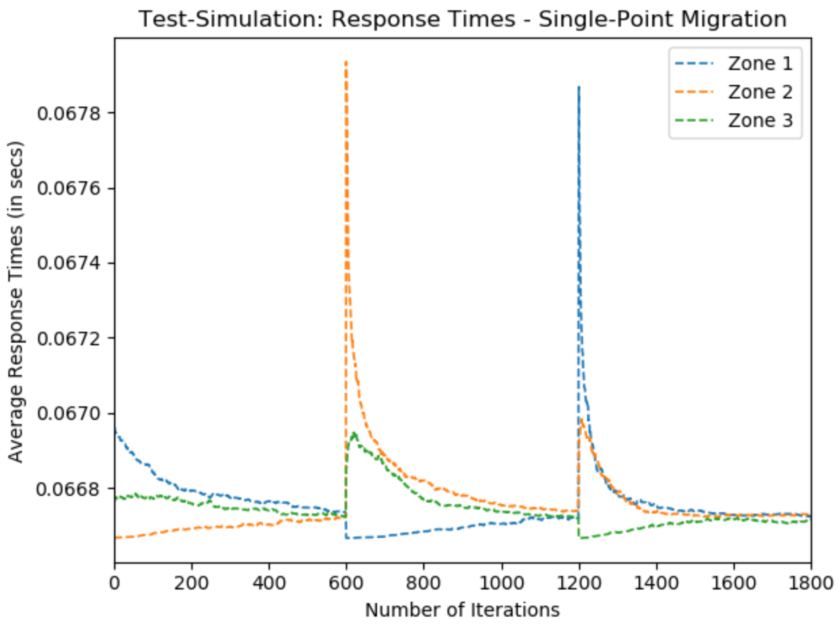
4.1.4 Results: Simulation of single-point migration

The final variation in the evolution algorithm implies the migration scheme where the VMs choose to migrate to the zones which are most affected by the request rates, i.e., the zone with highest avg. resp. time in a network. The process can be considered as the most simplest and natural selection of candidate zone as ideally a migrating VM should be considered to move to a affected zone by considering its network condition, without any need of other attributes to contemplate. The graph (Fig. 4.3a) for the particular migration variant shows network convergence with the static set of request rates in a simulation. The slopes representing three zones in the simulation showing the algorithm principle has been adequately being implemented. As it is shown in the static test-simulation graph, zones with higher request rates started to balance their response times first. It is because due to the zone with lowest response time (Z2) constantly making sharing its VMs to those two most affected zones (Z1 and Z3). The curve representing Z1 response time is very rounded or have a smooth arch which represents that the response time of this zone is getting equal to the rest of the connected zones in the network simulation. On the other hand, the Z3 curve has a small fluctuation at the beginning of the simulation as it started also to share its VMs to the Z1. When the Z1 and Z3 response time becomes almost equivalent, the migrating VMs in those zones stop moving from their corresponding zones. Rather these zones keep getting the VMs from Z2. The simulation (Fig. 4.4a) shows that the equilibrium of this network using single-point migration scheme takes quite long time. The reason behind is predicted as the number of VMs are migrating to affected zones using a traditional method (migration to highly affected zone in the network) takes a cost in the emergence behaviour. The choice of picking the most affected zone as a candidate without considering the situation of rest of the pool of VMs shows no significant target to address the network equilibria issue. The system becomes converged at almost 6000 iterations which is much longer than any other variants of the general algorithm.

The exactly similar behavioural pattern is shown when we introduced the dynamic request rates in the simulation for single-point migration. The graph in Fig. 4.4b represents such simulation. In the graph, the network converges almost at the same time with static requests (at 6000 iterations). In this set (set A), the avg. resp. time of all the zones shows almost simultaneous changes. This is a strange result then is shown in the static version. But at the second set of requests, the highly affected zones (Z2 and Z3) are showing the similar behaviour again, where after certain fluctuations both zones are equalising in a similar way. But the Z1 (with the lowest request rates) took the longest time to reach the converging point. At the third set, the results are much better when the most affected zones are almost equalised on an exact pattern and with about same response time. At 14000 iterations, the zone with moderate traffic (Z2) gains better response time average. Zone3 (with lowest avg. resp. time in last set C) shows the almost exact model as the previous sets' zones with lowest request rates.



(a) Static



(b) Dynamic

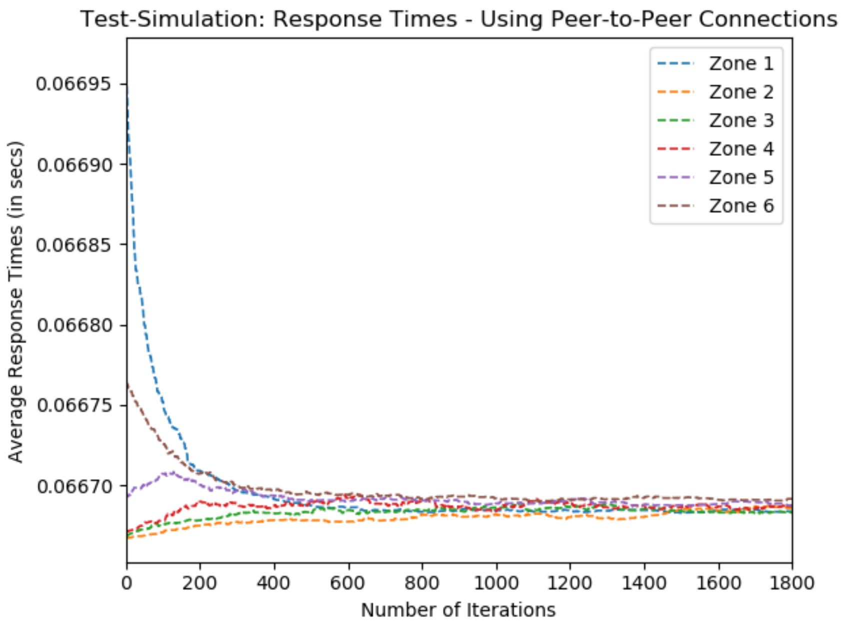
Figure 4.4: Simulation of Single Point Migration

4.1.5 Results: Simulation of uniform-site migration (informed) using peer-to-peer communication

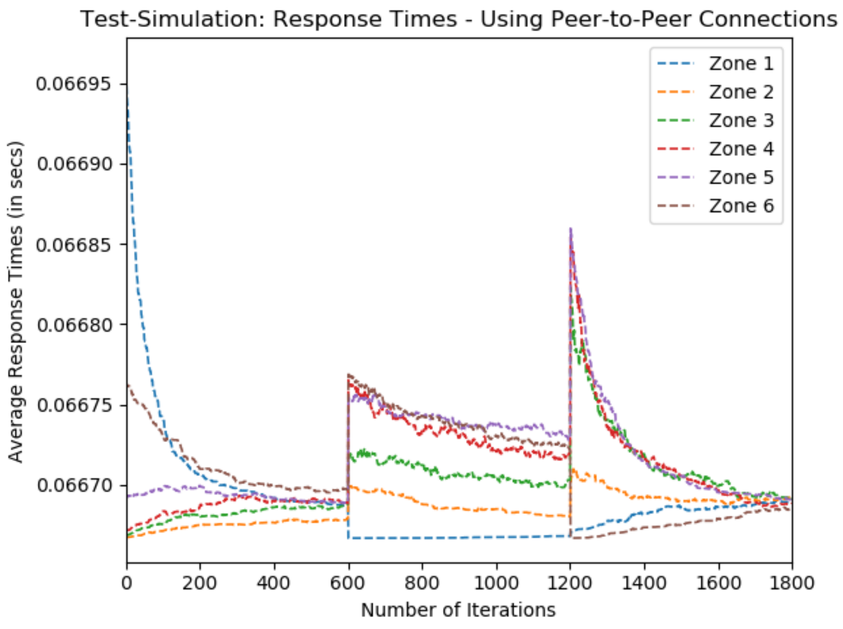
This test has a different approach than previous test-simulations. We administer this experiment with six zones. The zones are separated into clusters in the same topology as shown in Fig. 3.6. As we explained in Sec. 3.2.6, the migration VMs will only choose the candidate zones which is in the same clusters as their own zone. The graph/plot showing in Fig. 4.5a and 4.5b are representing the test results of the simulation with static and dynamic request rates to the zones respectively. Running the simulation with python several times we can observe that the system is converging in a very fluid curve. This means, all the zones in the network are converging in a simultaneous manner. This behaviour of the convergence gives us an interesting overview of how all the individual component of the system working together to achieve the same goal. In the simulation, our algorithm for uniform-site migration (informed) has been implemented to all the migrating VMs. Since the VMs have no inter-exchange relationship between other VMs the decision it makes to pick the next candidate zone is only influenced by the condition of the other zones in the same network (not necessarily in the same cluster in this case). The result with static request rates (Fig. 4.5b) exhibits a gradual increment/decrement of the every zone's avg. resp. time in a controlled simulation environment. Running the same simulation over and over again gives us an overview of all zone's emergence behaviour with our evolution algorithm design. It is noticeable that the zones which have the worst avg. resp. times are converging faster than the ones have lower avg. resp. times. Also at the same time, the zones with lower response times come to the converging point, i.e., the point in the graph (at almost 3000 iterations). Except for the Z2, all the zones are converged at some point around 11000 iterations. Z2 take much more longer times emerge with the other zones. Such behaviour of the Z2 was continuing to persist in all the individual experiments we conduct for this simulation. We will discuss and analyse this situation at the Chapter 5. In the rest of the simulation process, the response times in all the zones stay almost at the same value which depicts that the infrastructure stays stable till the rest of the simulation.

In the next figure (Fig. 4.5b), we present the test-simulation result for the uniform-site migration using peer-to-peer communication with dynamically changing request rates. In this test, the first set (set A) of request rates are injected into the zones until next set of the dynamic set. Naturally, the behaviour of the system is as same as the test with the static set. Zones are converged at the same point almost after the same iterations and Z2 (with the lowest request rate) takes longer to meet the convergence point. After 6000 iterations, the new set (set B) of dynamic request rates are introduced to the simulation. The system reacts to the new set in a strange manner. Here Z1 gets the lowest request rate and react very weakly to that. Zone 4,5 and 6 starts to converge slowly at the way and Zone 2 and 3 do the same together. It shows that the Zones are working as a group instead of emerging together as a whole system. The convergence period of this much longer than other periods. The zones do show a behaviour to slowly reaching towards the point but it seems like in this session or period the zones do not adjust their response times with the whole system. Changing the request rates at 12000 iterations the zones quickly shows fluctuations in response times and begin to adjust the system-wide avg. resp. time in an exact same pattern. This pattern is almost similar to the first set of the test except for this time the zones with heavier loads adjust their response time simultaneously and a fixed interval. Zone with the lowest amount of request rate (in this set Zone6) behaves in the similar fashion as previous tests.

We have tested this simulation with a longer period of time (18000 iterations) than other test-simulations. The system equalising period is longer with peer-to-peer communication topology and the number of zones is a big factor in this test. Undoubtedly, more zones mean the more VMs in the network. The main purpose of conducting this test is to inspect the network scenario with a higher number of system components and more complex topology. We increase the number of iterations to adjust the system according to long emerging period behaviours.



(a) Static



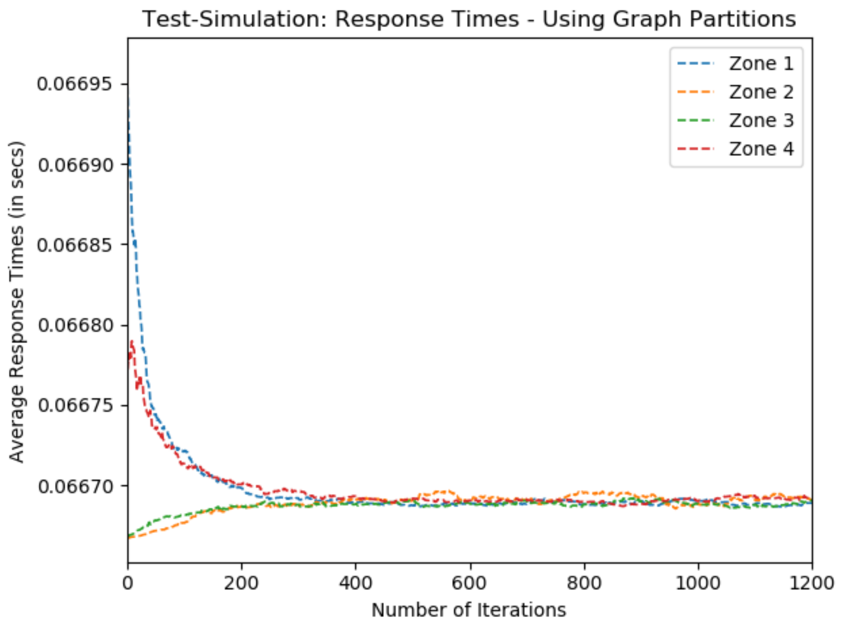
(b) Dynamic

Figure 4.5: Simulation of Uniform Site Migration using Peer-to-Peer Communication

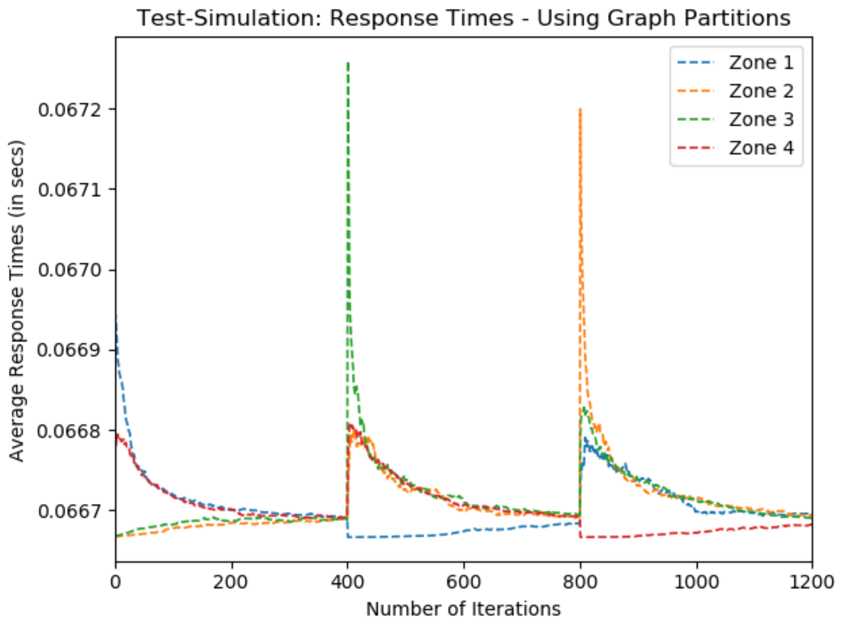
4.1.6 Results: Simulation of uniform-site migration (informed) using graph partitioning

In this test-simulation, we are experimenting the performance of our designed uniform-site migration (informed) algorithm on a scenario where all the zone in the network are divided into groups. Each group have only two zones and each zone is part of at least two groups. This scenario gives us a ring topology which is explained in Sec. 3.2.7. For this test, the number of zones is set to only 4 as we want to minimise the convergence period of the network. However, the number of VMs per zone is still the same as all other test-simulations. The total simulation time (in our case iterations) is also lowered to 12000 iterations. We start with the simulation first with the static set of request rates. Fig. 4.6a shows the graph that represents the results of this entire test-simulation. Analysing this graph we can observe that the zones (Z1 and 4) with higher rates started to stabilise their response time as they proceeded to receive migrating VMs from zones (Z2 and 3) with lower request rates. Faster convergence is perceived in this simulation. The curves/slopes representing avg. resp. time of zones with high/low traffic load (request rates) are decreased/increased in parallel respectively. The communication between all the zones are very sophisticated as the effect on a zone in one group will impact the game strategy of the other group that zone is connected as well. The graph showing that even though the groups have no domination on each other and the migrating VMs are taking decision independently and autonomously, the network can still show emerging behaviour as the strategy of the game (evolution algorithm) kicks in. Repetition of the same strategy over all the zones resulted in the same way which confirms that our algorithm is working efficiently by making migrating VMs pick the most affected zones as the candidate despite which group they belong too. Having the option to migrate to any zone of any group makes the migration procedure much efficient and productive when it comes to target a faster equilibria in the network avg. resp. time. Once the stabilisation takes place, the system becomes stable and keeps the condition as same during the rest of the simulation. The total converging time of the network (stabilising avg. resp. times in all the zones at the almost same level) is nearly close to 3500 iterations which is much faster than the simulation with naive uniform-site migration and single-point migration despite the fact that this test consists more zones and complicated scenario.

Then we implemented the same simulation, but this time we use the dynamically changing request rates set. In figure (Fig. 4.6b), all the zones converging at the same rate as the previous static rates' test. The simulation has the similar result as simulation experiment with uniform-site migration (informed) (Fig. 4.2b) and constantly equalising in the same fashion. The only difference is noticeable in this graph that the number of zones that are converging together is now 3 instead of only two due to the reason we have increased the number of zones in this test. The curve showing avg. resp. times for the highest request rates always coming down to convergence point in the exact same manner, i.e., the response times of all the zones are nearly the same with the second set of requests. This is because as in the first set (set A) the zones had an equal number of VMs in each of them but at the end of this period all the migrating VMs are already moved to the most affected zones. The VMs are distributed equally to all the affected zones. This is promising since it confirms that the variant of algorithm taking care of the situation and system reaching an equilibrium. As the second period starts with set B, the zone with the best response time is always started to send their own migrating VMs to the affected zones and so do the zones with moderate request rates share its VMs to the zones which need the most VMs. The randomness was removed from this algorithm and the VMs are given a better strategy to migrate. After the similar amount of iterations, the zones are coming to the same state every time the request rates are changing.



(a) Static



(b) Dynamic

Figure 4.6: Simulation of Uniform Site Migration using Graph Partition

The above test results confirm us that the implemented different variants of evolutionary algorithm are efficiently working and feasible to any given conditional scenarios. These tests represent the adaptness and capability of the algorithms to reach a target, in our case a network equilibrium. In the following sections, we will experiment the similar experiments with the proposed algorithms but in real-life test-bed scenarios with GCP and Google instances. The test-bed results can confirm us further about the performance of our algorithms and give us more insight on how the algorithm can actually affect the instances in the existing cloud systems.

4.2 Results: Test-Bed

In the test-bed scenario, the algorithms are put to the test in real-life cloud settings. The similar concept of implementing the different algorithms to check the network performance is carried out in this section. The initial experimental parameters are set for all the zones in same as follows:

```
The initial number of VMs for each zone = 7
```

This test will be set by adapting to GCP cloud by creating a number of zones with VMs. The following table shows the specifications for all the instance machine types running in the GCP data-centres in the initial step. This machine types can give us an overview of what is the capacity of each DC, which is similar to service rates of our test-simulation. Table. 4.2 shows the specifications for all type of instance we set up in the GCP:

Table 4.2: Instances Specifications for Test-Bed Scenario

Machine Name	**Machine Type**	**CPU**	**Memory**
lb-dc	n1-standard-2	2 vCPU	7.5GB
consul-master-dc	n1-standard-1	1 vCPU	3.75GB
nat-gateway-dc	n1-standard-1	1 vCPU	3.75GB
worker-dc	f1-micro	1 shared vCPU	0.6GB

We will address the zones as data-centres (**DC**) and migrating VMs as **worker-instances** for this test. To simulate the incoming HTTP request to the distributed web services we are using AB benchmarking tool as request generator. This entire setup and test infrastructure have been described thoroughly in the earlier sections (Sec. 3.3). The following table (Table. 4.3) gives our reader a glimpse of values and parameters set for test environment:

Table 4.3: Initial Parameters for Test-Bed Scenario

DC Name	**DC Location**	**Client Location**	**Number of Requests (-n)**	**Concurrency (-c)**
DC-1	europe-west3-c	europe-west3-a	999	700
DC-2	us-central1-a	us-central1-c	999	800
DC-3	southamerica-east1-b	southamerica-east1-a	999	600

All the clients are generating and sending the HTTP requests with AB tool with the same properties:

```
ab -n <no_of_requests> -c <no_of_multiple_requests_to_make_at_a_time> <server_domain_name>
```

The above commands create an imitation of HTTP requests incoming from different clients close to the geographical locations of different regional instance groups of the data-centres in the network. These clients will constantly create the same amount of requests to the web service URL which will be load-balanced to all the worker-instances which are serving as back-end servers of the load-balancers instances. Each worker-instances are providing a simple Apache web server as a part of distributed web service.

Data Collection Process

As we mentioned in the earlier section, we have given our web service a domain name to reach to the back-end servers with autonomously migrating instances. The requests to reach the web servers are sent to the worker-instances and replied back to the clients. The response times of the all the back-end servers are then measured in milliseconds and collected via the status report of the each data-centre's load-balancers. The load-balancing running in this instances is done by the HAProxy application which can provide the current real-time status of its back-end server. These status reports are valuable in this research work since we are going to make migrating decision in workers based on the current state of the data-centres. Since an lb-dc instance will forward all the HTTP request to the worker-instances it can be considered as the important point of data collection for each region. In each data-centre, the response time of each worker-instance is then summarised and divided by the number of total workers in that data-centre. This will give us the avg. resp. time of that data-centre at that point of time. A test script is created to calculate the data at an equal interval and collected for further analysing. The following part of data collection process gathers all the IP addresses of the lb-dc:

```
declare -a lb=('lb-dc1' 'lb-dc2' 'lb-dc3')
for argh in ${lb[@]}; do
lb_ip=$(gcloud compute instances list | grep $argh | awk '{print $5}')
done
```

The addresses are then used to gather HAProxy status for each DCs:

```
curl -sSL 'http://$lb_ip/haproxy?stats;csv;norefresh' | cut -d "," -f 2,61 | \
column -s, -t | grep worker | awk '{print \$2}'
```

Then every response times are collected, summarised and finally avg. resp. time are calculated.

```
avg_resp_time_all_dc=$(( sum_all_resp_time / no_of_workers ))
<dc>_resp_time=$(echo "$avg_resp / 1000" | bc -l)
<dc>_worker_time=$no_of_workers
```

This simple script (Appendix: F.1) dynamically finds all the ephemeral IP addresses of the lb-dc instances and finds all the worker-instances response time by reaching the HAProxy status page. Then all the response times are summed together and divided by the number of workers at that particular moment. HAProxy provides the response time status in milliseconds and we are converting into seconds for a better perspective. This script runs in intervals and appends the data to a CSV file.

Our test-bed includes comprehensive experiments by using various tools. The data collected as CSV format are projected in plots with the exponential weighted moving average (EWMA) to administer the performance evaluation of the applied algorithms. We discuss the individual test on our infrastructure in the cloud with the variant of algorithms in the following sections. A script (Appendix: F.2) to create a performance graph for each algorithm has been provided to create EWMA graphs with 100 window span. We are using the EWMA graphs to plot these tests as in the uncontrolled test as such can introduce a lot of noises in the plot. This is due to the reason that in the real-life scenario of the cloud environment the response time in the servers does not always gradually increases or decreases. The fluctuating nature of avg. resp. time will not any concrete projection of test results unless we narrow it down to moving average of the test instead of current data in every single point of the graph. A simple example of such problem is shown in Fig. 4.7, where we try to project the raw data extracted from a data-centre. The plot will also represent the raw data of avg.resp. time of the data-centre along with its EWMA.

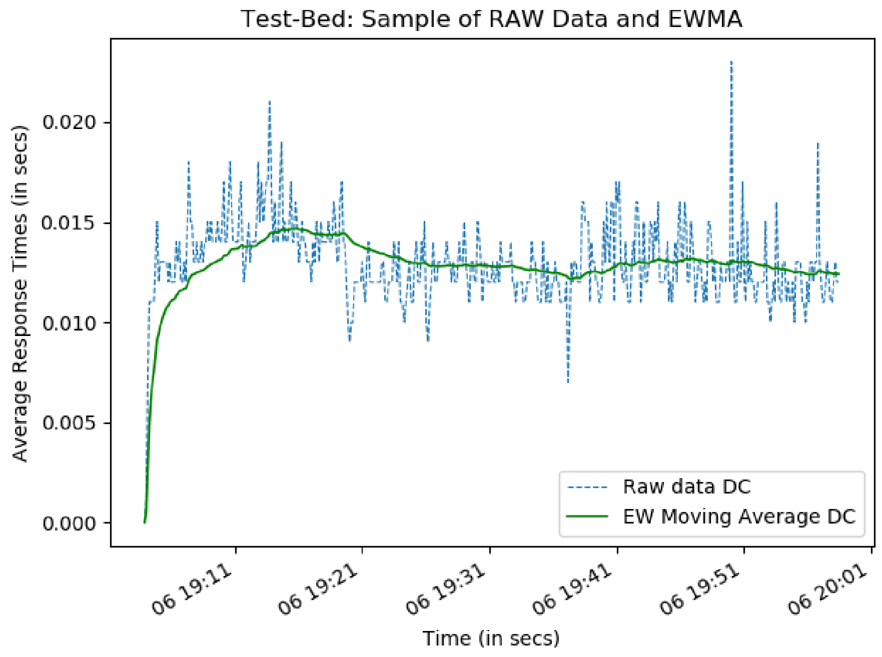


Figure 4.7: Test-Bed Sample of Raw Data and EWMA Graph of A Data-Centre

As we can see EWMA is a better presentation than of raw current data of the data-centre which can give us a better perspective of the algorithms' performance. The lines showing the EWMA is the better choice when it comes to project the entire systems' convergence period, instead of including all the raw data in the graph. In case of many data-centres in the network, we need to use a cleaner way to abstract the data for analysis purposes. In this graph (Fig. 4.7), the X and Y axis' are graphed as avg. resp. time (in seconds) of the DC and time (in the format: **Date Hour:Minute**) of testing respectively.

The scripts for collecting response time for all the data-centres in the network and saves it in the background. After a certain random interval (details are in Sec. 3.4.8) the migration procedure gets activated and picks a candidate data-centre based on the implied scheme. Some randomness are included for these operations activation since we are not allowing all the worker-instance to make the decision and migrate at the same time. To balance the number of instances in the back-end, we are letting the nodes become ready for migration in the different period of convergence. Based on the data gathered in each worker are used to make migration steps if the worker decides to move. If not migrating, the worker-instance uses the data for future migration procedure along with the newly updated data. If no migration happens the worker's migration procedure goes back to sleep and continue to run as the regular web server for its current data-centre.

4.2.1 Result: Test-bed with no migration

Before we start applying our proposed algorithms in the test-bed we like to find out how a cloud system reacts to the incoming HTTP requests when no migration algorithm is running in the worker-instances. We run a test-bed experiment, where all the web servers will be just response to incoming requests from corresponding nearby geo-located clients/users for the entire test period. Fig. 4.8 gives us the simple EWMA graph of this test result.

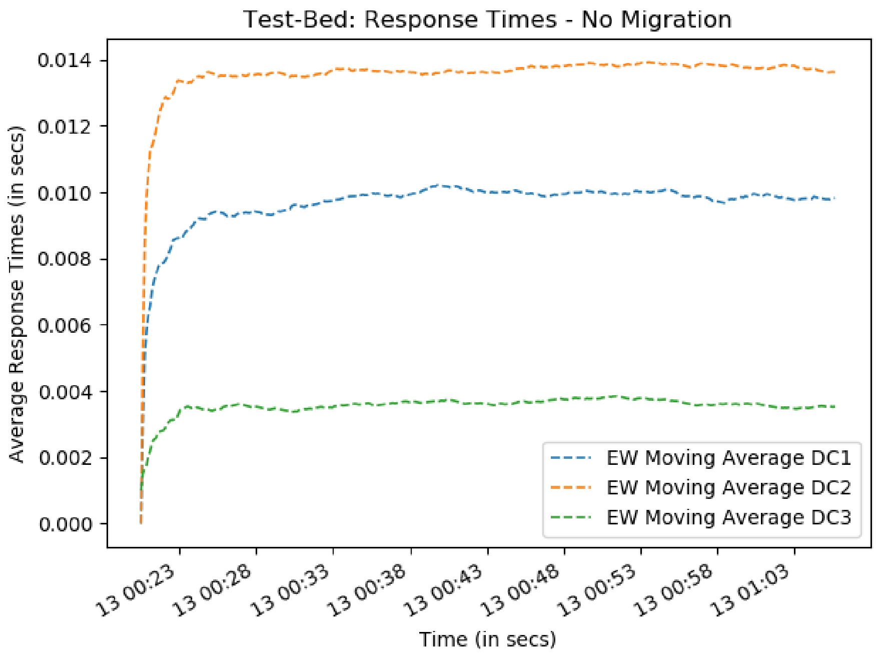


Figure 4.8: Test-Bed EWMA Graph for No Migration

It is clearly shown from this graph that the avg. resp. time have never gets changed on the data-centres as the number of user's requests increases to a specific area. Without an appropriate migration scheme, the regions with higher users (HTTP requests) tends to suffer more from QoS of web services as the number of users directly responsible for response time in data-centres. In the DCs with lower response time there can be many idle instances (that are not migrating) which can be ideally shared among the affected regions. Lack of communication of all the centres and absence of awareness of connected network conditions makes the system vulnerable and ineffective. A quick fix to this sort of scenario can be scaling the number of instances according to the requirement of the user's requests to maintain a suitable level of avg. resp. time. But as we mentioned in the introduction, this solution costs insignificant amount for the administrator of the large distributed web services based on a multi-regional cloud infrastructure. The graph (Fig. 4.8) gives a better confirmation of necessary of a suitable algorithm to address this issue in the cloud technology.

4.2.2 Result: Test-bed of uniform-site migration (naive)

This test-bed includes the implementation uniform-site migration (naive) in all the worker-instances across the network. In each data-centre, every worker nodes register itself as individual web services. However, as for migrating schemes, each worker will run the naive migration algorithm and adapt to the migrating principle accordingly. As the worker-instances in this algorithm choose in a completely random fashion despite the network condition, especially the condition of the most affected data-centre, the worker migrates can end up moving to any data-centre. Action as such can have an effect on the evolution procedure of the network as we can observe in Fig. 4.9. This particular figure represents the EWMA of avg.resp. time of the entire test-bed using naive uniform-site migration on the worker-instances. It is visible that the response time becomes very high in the early stage of the experiment as the HTTP requests starting to get distributed with the help of load-balancer to the back-end servers. In our test, the DC1, DC2 and DC3 getting a moderate, heavy and regular amount of traffic respectively (refer to Tab. 4.3).

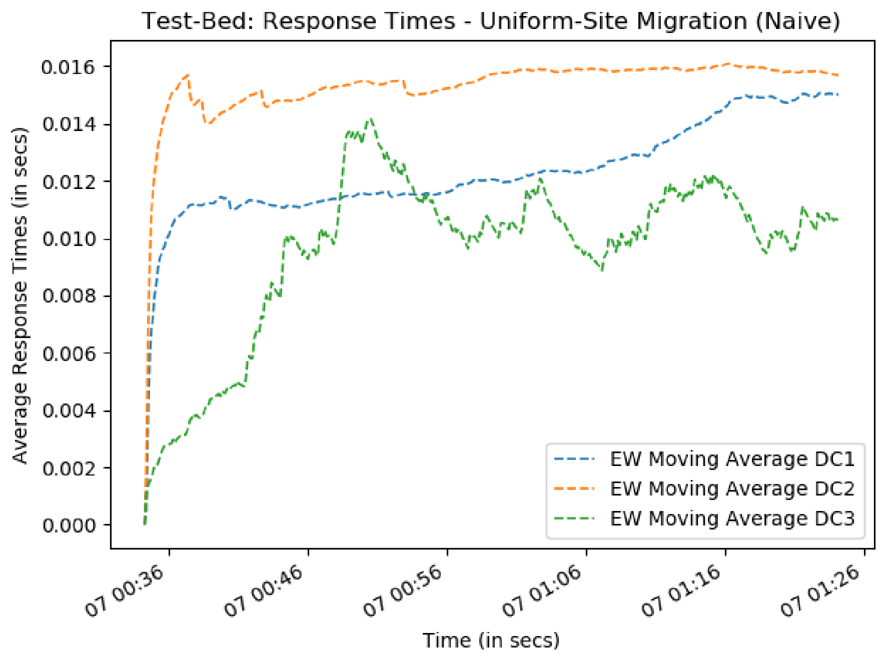


Figure 4.9: Test-Bed EWMA Graph for Uniform-Site Migration (Naive)

As the migration script gets enabled and VMs in each data-centre proceed to pick a random DC to migrate, the avg. resp. time in each data-centre starts to adjust the response time according to the number of worker-instances in their own region instance group. The algorithm does not have any particular view on what should be the strategy to reach a system goal, i.e., the convergence of the whole network response time. None of the DCs shows any emergence behaviour as we can see in Fig. 4.9. At the beginning of the experiment, DC1 and starts to get the most HTTP requests when DC2 gets a moderate amount. Even though the difference between the concurrency of the HTTP requests are very close to each other for all the clients, we should see little variance on the avg. resp. time accordingly. The system never really converge throughout the entire test period. We can see in some point the DC3 response time rises to the point where it meets the DC1 response time, but after a small period of time DC3 start to show uneven graph considering the other two data-centres. DC1 and DC2 also never becomes equal in response time until it reaches the end of the test-bed experiment. Even though it seems close to each other, running the scenario longer will show that the response time levels gets further from each other. All the workers in DCs migrating aimlessly in this test. This scenario shows no improvement on emergence behaviour in the infrastructure and the convergence of the system never really took place entirely. It might occur after a certain period of test, but it cannot be assured that this is certain in such case.

4.2.3 Result: Test-bed of uniform-site migration (informed)

The next test-bed experiment in GCP includes the implementation of uniform-site migration (informed) as the worker-instances migration principle. Similar to the above experiment, we spawn equal amount of worker-instances and apply this migration scheme. The test results are also included in a form of a graph where EWMA of avg. resp. time are shown. In the Fig. 4.10, we can see the avg. resp. time of DC3 takes the longest amount of time to reach the convergence point. DC1 and DC2 get equalised almost immediately after the test starts. These two data-centres have the highest number of incoming HTTP requests from clients. As the migration principle of instances is to move to another data-centre which has higher avg. resp. time its own data-centres, the workers can finally take better decision for migrating. This is the first time we can see a developing behaviour in the system. The data-centres shows a better relationship between them as the number of worker-instances are getting well-distributed among them. Since the data-centre DC3 with the lowest traffic loses its worker-instances the avg. resp. time slowly rises to its highest point. At the middle of this experiment, we can see the response time of DC3 rises to its highest pick and then starts to become again. At the same time, the DC1 starts showing a high pick in its own response time and it is assumed that it started to receive worker instances from DC3 and DC2. After a certain period, DC1 stabilises its own avg. resp. time and all three DCs become starting to show an equilibrium in the network. This is to consider as the convergence in the system has taken place.

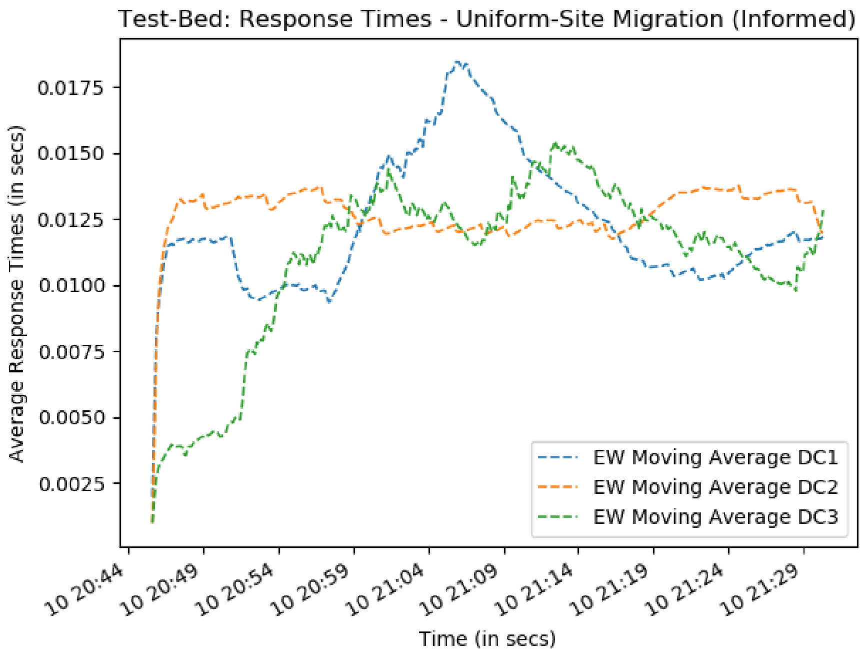


Figure 4.10: Test-Bed EWMA Graph for Uniform-Site Migration (Informed)

In addition, we like to point out that even thought the response times are tends to be fluctuate a little bit they quickly comes back to the same level and continues to show the same behaviour during the rest of the test. Randomness to pick the destination for workers when more than one DC is picked as candidate DC also takes an affect in the graph. This results the DCs' response time becomes irregular in some intervals if the workers moves to a less affected data-centre due to random moving. The random property has been eliminated from the algorithms in the next experiments. However, the algorithm perform quite well under such circumstances and always adjusted itself despite the irregularity. This is a very interesting progress on the general evolution algorithm which can be analysed further for better research purposes.

4.2.4 Result: Test-bed of biased migration

In this experiment, worker-instances are adapted to a better strategy than previous two variants of the algorithm. Not only they can pick a candidate data-centre observing the all

DCs network condition but also with a probability of moving to the destination DC. The worker-instances have a better awareness of the entire infrastructure and the decision that has been taken by all the instances collectively can/will influence the emergence behaviour all the individual components. Running this test with biased migration algorithm we can get the graph representing the results of the test-bed scenario. The graph is shown with EWMA curves in the Fig. 4.11.

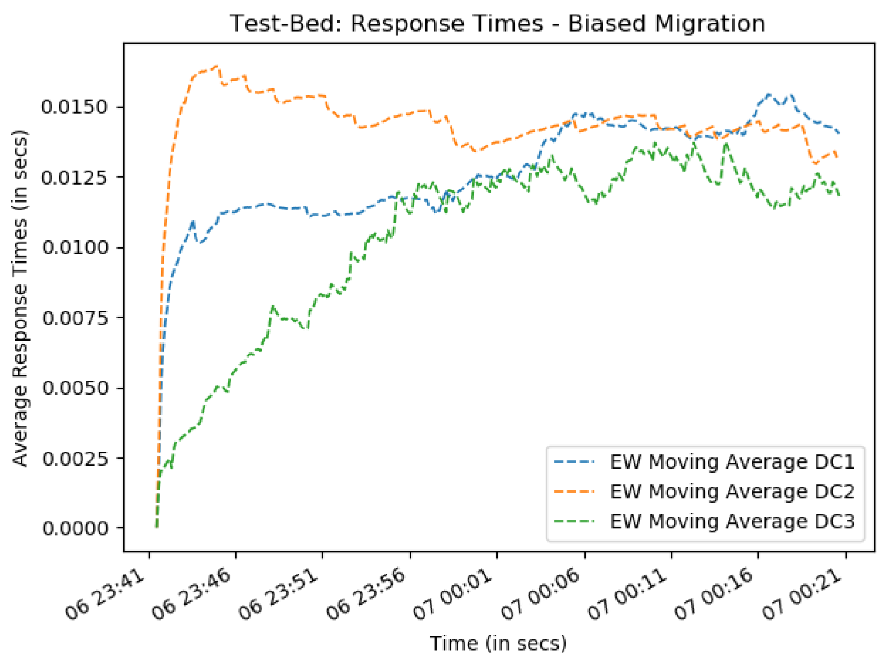


Figure 4.11: Test-Bed EWMA Graph for Biased Migration

In the graph, it is noticeable that the convergence period is almost the same as the test-bed experiments with uniform-site migrations (Sec. 4.2.2 and 4.2.3). The ability to pick a specific destination for migration, workers can now decide more efficiently and intelligently. At the beginning of the test, all the average response time eventually get higher according to their rate of client's HTTP requests. The instance groups then gathered the data from each data-centres and started to, calculate the probability to migrate to a destination data-centre which has the highest avg. resp. time for last certain period. When the migration mode gets activated in the worker-instances, they find the best probability and picks the data-centre which has the probability. Since the probability margins are really thin, migrating to a certain DC is also become very limited and precise. We can notice the response times in all the DCs are gradually coming to the same point instead of quick increase. The effect is due to the probability property that we introduced to the migration scheme. The DC with the highest number of requests (DC2) decreases the response time as it starts to receive migration instances from other two DCs. DC1 gets the moderate number of requests, so it keeps the avg. resp. time stable as it gains or loses instances. Finally, workers from DC3 started to migrate to either DC1 or DC2 (mostly DC2 as it has the highest response time) and response time in this data-centres increases due to lack of enough web servers in the back-end. This test shows a much better promising on not only the total emergence of the system but also the keeping the response time in the network in stable positions. Running the test for a similar amount to time gives a system-wide convergence at all the time. The fluctuations in the response time of all the data-centres are in very small amount and more steady than previous attempts.

4.2.5 Result: Test-bed of single-point migration (informed)

In this final test, we investigate our last variant of the algorithm (single-point migration) where the nodes/worker-instances migrates to next destination (data-centre) based on their avg. resp. times only. This is the general principle of VM migration where the instances are given a choice to pick the candidate data-centre in a natural selection, i.e., the data-centre with highest avg. resp. time gets the migrating instances. A similar investigation has been conducted in the GCP test-bed infrastructure but this time single-point migration scheme has been implemented on the workers. The results of this test have been graphed in Fig. 4.12 and performance of the algorithm has been analysed.

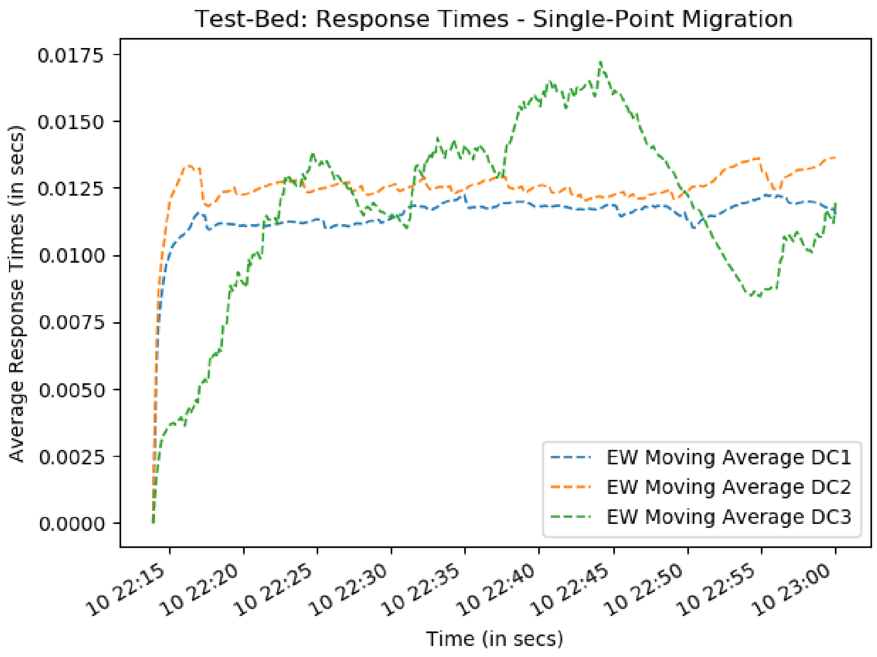


Figure 4.12: Test-Bed EWMA Graph for Single-Point Migration

All the DCs' response time are showing as EWMA plot in the graph. This algorithm shows a promising improvement in the convergence of the network as it is observed that all the data-centres stabilise their corresponding response time fairly quicker than every other algorithm (uniform-site (naive/informed) and biased) tests in previous sections. The migration scheme is quite straight-forward and prominent with any nature-inspired collective behaviour. As the migration operation proceeds, the data-centres with the lowest amount of incoming requests starts to gain on response times, since the migrating instances started to remove itself from those data-centre. When the instances get distributed to the heavy-traffic instance groups (in this case DC2), the avg. resp. time on these zones starts to get low. This procedure seems to be quicker in this test-bed. This can be assumed to be more effective since the calculation of the finding the candidate zone is fairly simpler for migrating worker-instances. But in the further part of the experiment, we can notice that the avg. resp. time of DC3 does get quite high at some point. This can be possible of quick dispersing of the back-end servers has an effect on the data-centre and this change need to get adjusted. In this test events, the DCs are failed to keep their avg. resp. time stable perfectly as test-bed with biased migration algorithm. However, the response time started to equalise itself with other nodes after a certain period and all the avg. resp. time of every DCs in the network become almost similar at the end of the experimental duration.

4.2.6 Result: Test-bed of migration schemes with dynamic HTTP requests

We extended our previous test-bed experiments with dynamically changing HTTP requests from the clients. This means, in this scenario the clients request to access the web service in the specific geographical area will increase/decrease in the mid-experiment. Tab. 4.4 shows the request rate sets we have implemented for each client individually. With this test scenario, we like to observe the performance of the variants of migration algorithms when the number of users/clients changes dynamically in different nearby regional locations. We need to investigate how our migrations schemes adjust the number of idle instances in the data-centre autonomously in order to balance the overall network response times. The primary goal of this tests is to evaluate the proposed variants of the algorithm in the dynamically changing environment for a better perspective of the algorithms' performance.

Table 4.4: Dynamic HTTP Request Sets in Clients for Test-Bed Scenarios

	Set A		**Set B**	
Client Location	**Number of Requests (-n)**	**Concurrency (-c)**	**Number of Requests (-n)**	**Concurrency (-c)**
europe-west3-a	999	700	999	600
us-central1-c	999	800	999	700
southamerica-east1-a	999	600	999	800

We start by sending HTTP requests (with AB tool) with the first set of clients (set A) towards the web service domain name. The graph (Fig. 4.13) represents the EWMA graph for the test-bed scenario with uniform-site migration (naive) when the number of clients requests are changing. We have changed the set A to B in the middle of the experiment. In the graph, we can observe that the network never converges throughout the first set session which is similar to the test-bed scenario in Sec. 4.2.2. After changing the clients' request rates the system seems to be starting to get stable but we assume that behaviour is still random due to the strategy-less approach of the algorithm. The avg. resp. times of all the data-centres are in the same level at the beginning of set B, which makes response times keep in almost the similar level as the difference between the concurrency rate from clients is very small and almost indistinguishable for the data-centres.

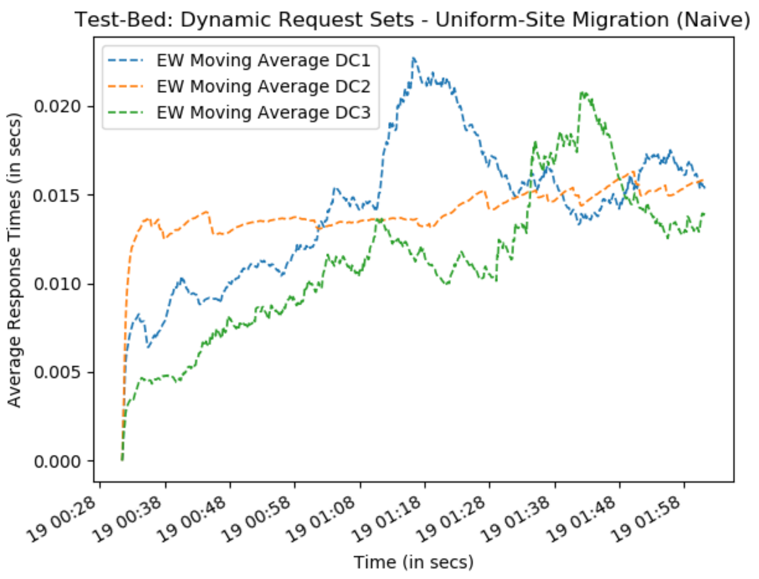


Figure 4.13: Test-Bed EWMA Graph for Uniform-Site Migration (Naive) with Dynamic Request Sets

Implementing the similar experiment with the uniform-site migration (informed), we can evaluate the performance of this scheme in a dynamic HTTP request rates. Fig. 4.14 showing the EWMA graph for this particular test scenario where the system showing emerging behaviour in a similar way as previously when the set A session is running. The period for converging (point in 02:38 of the Fig. 4.14) not as fast as biased and single-point migration schemes but it is showing some sort of equilibrium in avg. resp. time of the all the connected data-centres. After changing the request rates set (set B) of the clients we starting to notice the instability in avg. resp. time in the data-centres. This means the response times are reacting to the sudden change of the request rates. Since the response time level are almost similar in all the DCs in this session the system showing quicker convergence period (around 03:23 in the graph) as it proceeds.

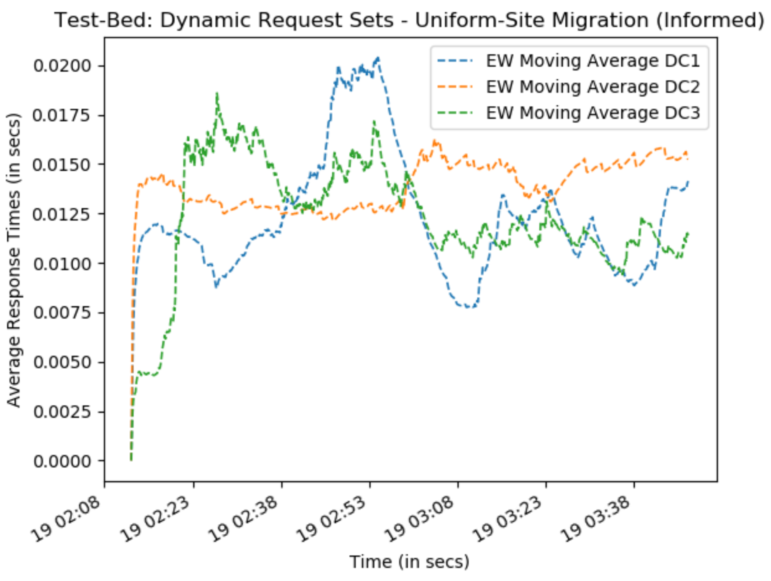


Figure 4.14: Test-Bed EWMA Graph for Uniform-Site Migration (Informed) with Dynamic Request Sets

Now we evaluate the similar test-scenario with the biased migration scheme of the evolutionary algorithm. We are expecting the identical results as before in the first set session of the test-bed. We like to point it out that these tests might not look exactly the same due to the facts that these are completely new test-bed results and the network overhead can influence greatly while performing the experiment and gathering numerical data. In the figure (Fig. 4.15) representing the dynamic request rate changes in the test-bed scenario with the biased migration scheme implementation. This algorithm gave a good performance at the test-bed experiment with single set request rates.

We are noticing the similar results in this case as well especially when it comes to stabilising the network by balancing the avg. resp. time in all the data-centres and fast converging period of the system. We can observe that at the second session of the test scenario, data-centres are stabilising their response time as the migrating instances are choosing the next destination with the probability. The graph shows that the emerging slopes are more gradual with this migration scheme which is also expected in the experiment. The system holds the avg. resp. times at both sessions of this particular test-bed in very stable positions. Even after the sets are changed the system does fluctuate from its stabilised position but quickly becomes stable again at another point after a certain period. Due to the network/protocol overhead, this sort of behaviour is normal for any distributed web servers. Here, we particularly investigating how the scheme is bringing the system back to a stabilised stage by observing the network condition in a specific moment.

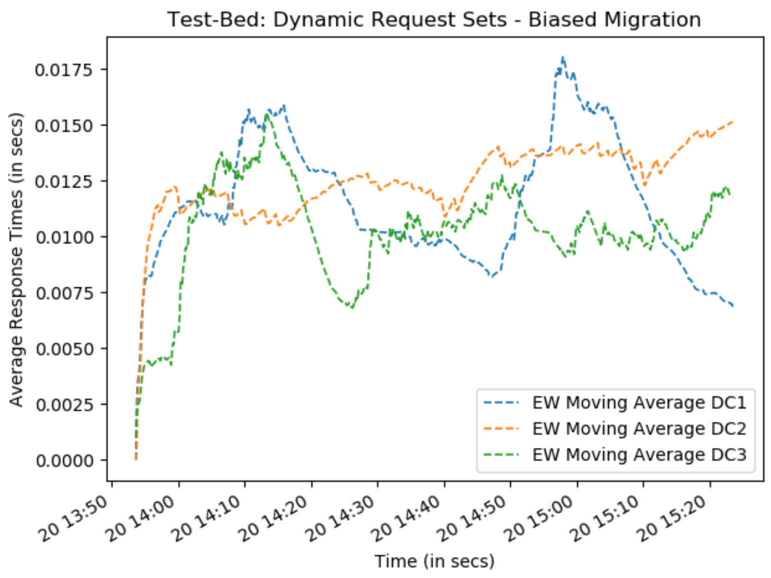


Figure 4.15: Test-Bed EWMA Graph for Biased Migration with Dynamic Request Sets

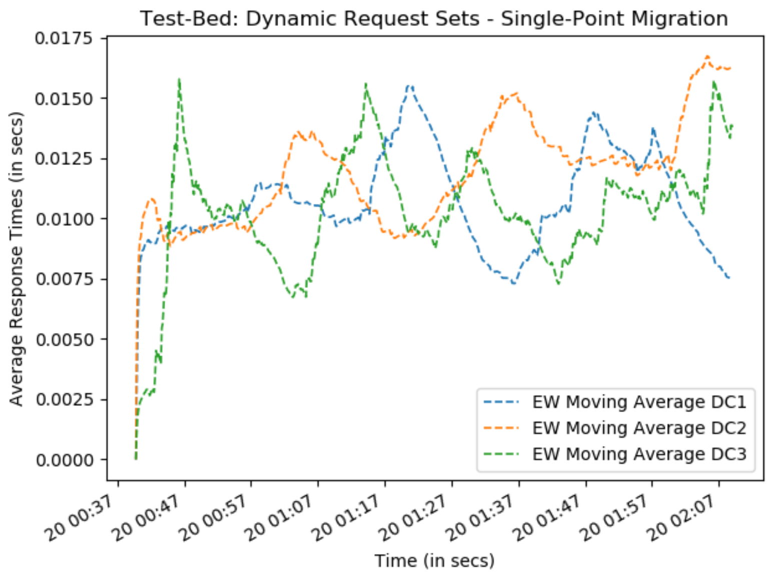


Figure 4.16: Test-Bed EWMA Graph for Single-Point Migration with Dynamic Request Sets

Finally, we have executed the experiment with single-point migration script with dynamic client request sets. Fig. 4.16 giving us the graph with EWMA plots with the collected results. The graphs show us how our migrating instances are handling the sudden change of the number of clients (or incoming HTTP requests) dynamically in the midpoint of the test-bed. The avg. resp. time is considerably adjusting itself as the same way in the second session with set B as it was in the first session with set A of the Tab. 4.4. The system has some fluctuations in the response time but it shows a good emergence behaviour during the test period. Avg. resp. time of all the data-centres seems to be continuously stabilising and the convergence period of the system are same as fast as the tests with a single set of client request rate. Since we do not have any probability in this scheme as we had for biased migration, the migrating instances move more frequently in this test. We can notice the slope representing avg. resp. times are showing slightly more fluctuations than biased migration graph (Fig. 4.15) with dynamic client request sets.

In the next chapter, we give a comparison of our simulation and real-life experiment results in details. The thorough discussion will be conducted to give our readers a conclusive understanding of the proposed algorithms' performance in the exhaustive test scenarios.

Chapter 5

Discussion

Every small factor of an individual experimental result can give a better overview of a bigger outline. In the chapter, we discuss the organised experiments in controlled and real-life (unregulated) scenarios that we have produced in the previous chapter. Our proposed algorithms were implemented and executed in multiple different scenarios which give are exceptional results from the tests. These results are then collected and projected in plots to readers to have a better perception of the performance of the algorithms. The performance graphs are then evaluated and compared to each other to find the best possible solution to meet our project goal. Comparison of test results gathered from simulations and real-life test-bed can give us a good recognition of how our algorithms perform in a controlled theoretical environment and in the real cloud system. This is to prove the concept of autonomous VM migration in cloud computing while maintaining a QoS of distributed web service. The comprehensive review of our algorithms is beneficial for visualising the accomplishment of the algorithms in various challenging events. We are comparing the variants uniform-site (informed/naive), biased and single-point migration operation in the test settings.

5.1 Discussion: Uniform-Site Migration (Naive)

The experimental numerical results of both in simulation and real-life are plotted to graphs and compared thoroughly as we proceed to discuss the performance evaluation of the uniform-site migration (informed). The algorithm has no particular strategy to reach a goal or in our case meet the evolutionary game theory requirements. Since all the migrating VMs moving to any zone/region completely randomly the system/network shows an incompatibility to balance the avg. resp. time. between the data-centres. Random movement of the workers in the data-centres tends to have a serious effect on converging the network as a whole. This is projected in both simulation and real-life experiments. In the simulation setting, the system does become stable at a point but the procedure took a significant amount of time. We tested the same simulation several times (approximately 100 times) and collect the average of all those simulations. The result seems to be giving an unstable system without a meaningful target. The purpose of designing this algorithm was to test a method where no strategy is introduced to the components. The same behaviour is noticeable in the test-bed experiment in the cloud environment where the number of worker-instances is not appropriately distributed among the data-centres, especially those with the highest amount of HTTP requests. The migration procedure of workers in this test shows no collective behaviour and the network never gets converged at any point. To consider an evolutionary game theory to succeed, it is recommended that every player/component in the game have similar strategy without knowing the condition of other players. This strategy should be designed to reach a goal in incrementally instead of jump to the solution in one step. The continuous and repetition of such emerging behaviour conclude in a system-wide steadiness and the entire system later become stable collectively. We do not see this behaviour in the test simulation with the naive algorithm. Neither we can observe any improvement of the scheme adaptation in the real-life situation. In the controlled environment, the infrastructure does become stable at one point but that result is not constant in every single experiment. In the other hand, test-bed scenarios never stabilise the response times in any number of experiments. This is to prove that a evolutionary game theory can only beneficial when a strategic approach is implemented, i.e., the components in the setup have a specific plan. The algorithm is then later updated to apply a better method to solve the performance issue of the proposed evolutionary game theory.

5.2 Discussion: Uniform-Site Migration (Informed)

Same as previous discussion, we compare the the results collected for the uniform-site migration with informed properties from the simulation and test-bed settings. The graphs created from both environments, shows us better result than the previous algorithm test. Since, in this test the migrating instances are given a choice between data-centres those who have worse response time than their currently allocated data-centres, the instances makes a better decision to pick a candidate to migrate. The performance improvement of this variant of algorithm is clearly visible in the graphs. Comparing the graphs (simulation vs test-bed), the nodes tends to be shared or distributed among all the DCs more competently. In the controlled test, the nodes are making a quick decision to choose the more affected data-centres which stabilises the avg. resp. time of those affected data-centres, since the number of back-end servers are directly involved for decreasing the response times. Similar results can be shown in the test-bed scenarios, as the entire system converges faster when instances can have a better decision making skill. The stabilisation of avg. resp. time in all the data-centres (network-wide) improving the performance of the test as the objective of game theory is met. This particular algorithm has a better accomplishment to reach our project goal than of with naive scheme implementation. The algorithm is designed to prove the concept of introducing a behavioural attributes in each component in the game to approach a intelligent strategy to maintain the system's condition. With such property, an individual component (in our case migrating instances) can contribute towards a better result, which is also beneficial for all other components in the same system. As we discussed, the convergence behaviour is present in this algorithm test and it is faster than of naive uniform-site migration tests. Especially in the test-bed scenario, the worker-instances shows a emergence behaviour to reach the convergence level and it is persistent throughout the rest of the investigation. We like to point out that the test-bed result is not exactly equivalent to the controlled (simulation) result. The theoretical analysis of the algorithm in the simulation is done in a system where no outside noise or error is introduced, wherein the real-life experiments the web servers are bound to be affected by the network congestion, time-out, network connectivity etc. Considering the randomness of network condition it is interesting to see how the algorithm still manage to control the system stabilisation as one whole connected entity and maintaining the QoS of the web servers at the same time.

5.3 Discussion: Biased Migration

The test results for biased migration tests gives us a better overview the efficiency of our evolutionary algorithm. Not only this migration scheme take the avg. resp. time of data-centres into account but also implies a probability based migration property which makes the convergence take place in a very steady method. The purpose of this migration application is to take the entire network's condition into account while making a migrating decision. This process makes the worker-instances take better choices to target affected data-centres. The graph representing the simulation results gives us a good overview how quickly the convergence in the system took place and how stable the system is throughout the rest of the test process. This means the response times are balanced equally faster with this migration algorithm than of other previously mentioned algorithms. The similar test results are observed in the test-bed scenario, where all the response time in data-centres came to the convergence point fairly quicker than previous ones. The stabilisation of the avg. resp. time in all the data-centres makes us confirm that biased migration scheme application in the workers has a better strategy when it comes to not only to system work together but also keep the equilibrium intact the whole time. Having a dynamic environment in the test-bed, the system tends to deflect from its own position, but with an effective design, the system can autonomously keep it balanced without any help of an external controller. Repeating the same process constantly makes the system more aware of the current network situation and since all the worker-instances can keep a track of last few network condition, minor fluctuations will not make any major difference in the stability of the system. We confirm from both the test-bed and test-simulations results that the condition of the infrastructure shows better performance when we adopt the biased migration.

5.4 Discussion: Single-Point Migration

In this final test, we are investigating the quality of the single-point migration algorithm on an infrastructure. Similar to the previous test, both scenarios such as simulation of such test and real-life utilisation on test-bed in cloud system GCP have been organised. The test results are set in graphs where we can analyse the collected data from the experiments. As we can inspect in the simulation the system does become stable quick as the biased migration but it had a hard time keeping the network stable. The avg. resp. time of the data-centres shows a small number of fluctuations as it proceeds which is also noticeable in the test-bed experiment. The convergence point where all the response times are becoming at the same level happened fairly quickly. This has the similar pattern as biased migration method. But the system tends to become unstable in cloud system as it continues. The algorithm only considers the data-centres with the highest amount of avg. resp. time as the migration candidate which might not take into account as other nearly affected data-centres with high response time. This makes the algorithm blind-sighted as it proceeds to pick the candidates. The effect of single-point migration algorithm's is still perceived in the graphs, as it has the better efficiency to keep the system stable during the test-bench but the result is not as perfect as the instance deployment with biased migration. The test-bed still managed to keep the QoS of the web server in its best quality during the test and cloud infrastructure gives a moderate and settled performance with this variant of the algorithm.

Table 5.1: Algorithms Performance Chart in Test-Bed Scenarios

Algorithms		**Uniform-Site (Naive)**	**Uniform-Site (Informed)**	**Biased**	**Single-Point**
Complexity		Simple	Slightly Moderate	Hard	Moderate
Test-Simulation Performance	**Convergence Period**	Slow	Fairly Quick	Fast	Fast
	Resp. Time Stability	Not Very Stable	Fairly Stable	Very Stable	Fairly Stable
	Overall Performance	Good	Good	Best	Better
Test-Bed Performance	**Convergence Period**	No Convergence	Fairly Quick	Fast	Fast
	Resp. Time Stability	Not Very Stable	Fairly Stable	Very Stable	Fairly Stable
	Overall Performance	Weak	Good	Best	Better

To give a complete picture of test performance of the different variants of the algorithm we have created a table (Tab. 4.3) which will give our readers a brief overview. We categorised the performance scales in different levels and commented on the achievement of the schemes by implementing them in cloud test-bed system and the simulation scenarios.

The classification of experimental results are marked as levels gained by all the variants in their accomplishment level on how much they influenced the network is controlled and real-life setup in terms of complexity level of designing and implementing the algorithms in test scenarios, how well-balanced the response times are in the entire network, how fast the system converges and their overall performance.

By ranking the effectiveness of the algorithm for worker-instances to choose a data-centre (which needs the most back-end servers) to pick as migrating destination and capacity of making the system converge in the fastest manner from any given condition in the network, we consider biased migration algorithm is the finest variant of the proposed evolutionary algorithm. Not only this algorithm makes the system (cloud infrastructure) stabilises faster but also it can keep that feature for longest (or in our case rest of the test procedure). Next to the performance criteria, with single-point algorithm system also shows an outstanding result. The algorithm proves to stabilise the cloud infrastructure at a similar pace as biased migration test but fails to maintain the position as good. It still shows gives us a good result when it comes to reaching the goal to prove the concept that an autonomous system can be as capable as a system with centralised control feature. The naive uniform-site migration fails to converge the system for the lack of any strategy in the algorithm. We overcome this deficiency by improving the emerging behaviour of migrating nodes/instances by proposing the informed uniform-site migration with additional characteristic/feature such as choosing candidate data-centre which is more distressed due to higher avg. resp. time. All the variants of algorithm show that an autonomous system can be adapted in cloud services in order to cut back on high maintenance/service of the web servers and cost of instances by distributing idle VMs in a more productive way using evolutionary game theory.

Chapter 6

Conclusions and Future Work

In this emerging world of cloud computing with various web services, designing an autonomous and self-management characteristics for a system with services is a complex method. Not only a developer needs to construct the architecture of the design but also has to implement an appropriate algorithm that can co-operate with that system. The functionality of a VM/instance in a cloud framework is very much depended on connectivity, network stability, service discovery, response time, QoS and many other features of the communicating network. The study of web service developing is continually growing, as these services have the advantage of easy deployment for a number of different applications. A VM with features such as autonomy, self-organising property, network condition awareness and most of all the self-capability of independently migrating to any data-centre (that is in need of more VMs), sharing idle web servers between data-centres and to reduce costs of instances in a cloud system plays a vital role in the cloud service infrastructure. By using an adequate strategy and well-fitted algorithm VMs can take a better decision and change the network condition without any help of additional controller system.

6.1 Conclusion

We studied the concept of developing an algorithm by adopting evolutionary game theory and embed the design to the VMs to introduce independent behaviour of migrating with an adequate approach. Our project goal is to implement a suitable algorithm which can balance the avg. resp. time in a cloud system by reducing scaling of additional VMs and utilising the idle VMs by sharing them among the data-centres. Our evolutionary algorithm has a better perspective of the entire network condition and takes a decision individually and in a calculative manner to migrate to another affected zone if the avg. resp. times get higher due to high HTTP user requests towards the web services. We designed this algorithm to increase the better utilisation of non-operative or idle VMs in a better manner and enforce a collective behaviour in them. The proposed algorithm adapts the evolutionary game theory to achieve such characteristics in the VMs which can provide us with a cooperative, self-reliable and advanced inter-network relationship in the cloud. The algorithm is then abstracted into four distinctive variants and inspected in extensive evaluation processes. The results are then compared to each other and we get an in-depth outcome of the attainment of our proposed algorithms. Out of four distinct variants of algorithms, we have concluded that biased migration scheme performs the best in both controlled and realistic environment. Our proposed evolutionary algorithm shows that using this algorithm the system can reach an equilibrium as every single associated component in the same network can contribute towards a specific goal to reach a convergence level in that system. Implementing these algorithms with evolutionary game theory in the VMs, we can make them more autonomous and completely independent of any centralised system while maintaining the QoS of distributed web services.

6.2 Contribution

The following gives a quick overview of our contribution to this masters thesis project:

- We proposed an evolutionary algorithm to implement in a VM to introduce autonomous behaviour in the system. This algorithm can help VM analyse network condition of all the regions in the infrastructure and migrate to affected ones to equalise the response time.
- We implemented four variants of algorithms to improve the strategic decision to migrate and tackle the system-wide network stability.
- These variants are then implemented in simulations to asses the efficiency of the algorithms. We tested one of the variants in the different topological scenario and observe the performance.
- The variants are also implemented in numerous test-bed scenarios in the cloud infrastructure to produce real-life scenarios and gathered numerical data evaluate the success of the designed principle.
- Finally, we summarise by extracting the data collected and put them in the graph models. We compare the graphs among simulation and real-life to give our readers of this thesis a detailed perception of proposed evolutionary algorithm evaluation. In addition, we evaluate the performance of the entire network along with the migration schemes implementation on the VMs, which gives us a thorough overview of our working algorithm's performance in any given condition.

6.3 Future Work

We include several potential work that can be investigated/developed in the future. These potential work are mentioned below:

- A better alternative can be implemented to activate the migration schemes in the worker-instances while testing, instead of using only a simple random wake-up and execute features.
- More comprehensive review can be conducted with high intensive exhaustive test-scenarios and a complex infrastructure to evaluate the algorithms' performance.
- A new algorithm can be designed to observe and predict the network condition in the preliminary stage and initiate the intermediate stage to make a better migration choices in advance.
- A test with each data-centre equipped with different variants of the algorithm to observe how the system setup performs can be an interesting experiment.

Bibliography

[1] R. Jain. *"The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation, and Modeling"*. John Wiley and Sons, Inc., New York, Apr. 1991.

[2] M. D. Neto (IBM). *"A brief history of cloud computing"*. [online] Available at: https://www.ibm.com/blogs/cloud-computing/2014/03/a-brief-history-of-cloud-computing-3/.

[3] Techopedia. *"Virtual Machine Migration (VM Migration)"*. [online] Available at: https://www.techopedia.com/definition/15033/virtual-machine-migration-vm-migration.

[4] Pluralsight. *"What is Hypervisor?"*. [online] Available at: https://www.pluralsight.com/blog/it-ops/what-is-hypervisor.

[5] S. Meier. *"IBM Systems Virtualization: Servers, Storage, and Software"*. International Business Machines Corporation, pages (2, 15–20), Apr. 2008.

[6] P. Viswanathan (Lifewire). *"Cloud Computing and Is it Really All That Beneficial?"*. [online] Available at: https://www.lifewire.com/cloud-computing-explained-2373125.

[7] Google. *"Setting Up HTTP(S) Load Balancing"*. [online] Available: https://cloud.google.com/compute/docs/load-balancing/http/.

[8] Python Software Foundation. *"Python"*. [online] Available: at https://www.python.org.

[9] Mitchell Anicas. *"An Introduction to HAProxy and Load Balancing Concepts"*. [online] Available at: https://www.digitalocean.com/community/tutorials/an-introduction-to-haproxy-and-load-balancing-concepts.

[10] J. Yue, B. Yang, C. Chen, and X. Guan. *"Chasing the Most Popular Video: An Evolutionary Video Clip Selection"*. IEEE Communications Letters, pages (781–784), May. 2014.

[11] J. Zhang, F. Dong, D. Shel, and J. Luo. *"Game Theory Based Dynamic Resource Allocation for Hybrid Environment with Cloud and Big Data Application"*. IEEE International Conference on Systems, Man, and Cybernetics, Oct. 2014.

[12] D. Niyato and E. Hossain. *"Dynamics of Network Selection in Heterogeneous Wireless Networks: An Evolutionary Game Approach"*. IEEE Transactions on Vehicular Technology, 58(4):(2008–2017), May. 2009.

[13] J. Zhang, W. Xia, Z. Cheng, Q. Zou, B. Huang, F. Shen, F. Yan, and L. Shen. *"An Evolutionary Game for Joint Wireless and Cloud Resource Allocation in Mobile Edge Computing"*. 2017 9th International Conference on Wireless Communications and Signal Processing (WCSP), Dec. 2017.

[14] C. Clark, K. Fraser, S. Hand, J. G. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield. *"Live Migration of Virtual Machines"*. USENIX Association, 2nd Symposium on Networked Systems Design & Implementation, pages (273–286), May. 2005.

[15] M. Duggan, J. Duggan, E. Howley, and E. Barrett. *"An Autonomous Network Aware VM Migration Strategy in Cloud Data Centres (ICCAC)"*. 2016 International Conference on Cloud and Autonomic Computing, Dec. 2016.

[16] H. W. Choi, A. Sohn, H. Kwak, and K. Chung. *"Enabling Scalable Cloud Infrastructure using Autonomous VM Migration"*. 2012 IEEE 14th International Conference on International Conference on High Performance Computing and Communications, Oct. 2012.

[17] S. B. Melhem, A. Agarwal, N. Goel, and M. Zaman. *"Markov Prediction Model for Host Load Detection and VM Placement in Live Migration"*. IEEE Access, 6:(7190 – 7205), Dec. 2017.

[18] S. Nanda and T. J. Hacker. *"TAG: Traffic-Aware Global Live Migration to Enhance User Experience of Cloud Applications"*. 2017 IEEE International Conference on Cloud Computing Technology and Science (CloudCom), Dec. 2017.

[19] J. Son, A. V. Dastjerdi, R. N. Calheiros, X. Ji, Y. Yoon, and R. Buyya. *"Cloudsimsdn: Modeling and Simulation of Software-Defined Cloud Data Centers"*. 2015 15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid, page (475–484), Jul. 2015.

[20] V. Kherbache, É. Madelaine, and F. Hermenier. *"Scheduling Live Migration of Virtual Machines"*. IEEE Transactions on Cloud Computing, 99:(1–1), Sep. 2017.

[21] F. Rodríguez-Haro, F. Freitag, and L. Navarro. *"Autonomous Management in Virtual-machine-based Resource Providers"*. Second IEEE International Conference on Self-Adaptive and Self-Organizing Systems, Oct. 2008.

[22] S. E. Benbrahim, A. Quintero, and M. Bellaiche. *"New Distributed Approach for an Autonomous Dynamic Management of Interdependent Virtual Machines"*. 2014 8th Asia Modelling Symposium (AMS), pages (193–196), Sep. 2014.

[23] J. Rao, X. Bu, C.Z. Xu, and K. Wang. *"A Distributed Self-Learning Approach for Elastic Provisioning of Virtualized Cloud Resources"*. 2011 IEEE 19th International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems, Aug. 2011.

[24] P. T. Endo, M. S. Batista, G. E. Gonçalves, M. Rodrigues, D. Sadok, J. Kelner, A. Sefidcon, and F. Wuhib. *"Self-organizing strategies for resource management in Cloud Computing: State-of-the-art and challenges"*. 2nd IEEE Latin American Conference on Cloud Computing and Communications (LatinCloud), Jun. 2014.

[25] D.J.T. Sumpter. *"The Principles of Collective Animal Behaviour"*. Philosophical Transactions of the Royal Society B: Biological Sciences, Feb. 2018.

[26] D.G. Harper. *"Competitive Foraging in Mallards: 'Ideal Free' Ducks"*. Anim. Behav., 30:(575–585), 1982.

[27] J. M. Smith and G. R. Price. *"The Logic of Animal Conflict"*. Nature Publishing Group SN, 246:(15–18), Nov. 1973.

[28] T. L. Vincent. *"Evolutionary Game Theory, Natural Selection, and Darwinian Dynamics"*. Cambridge University Press, pages (72–87), Aug. 2009.

[29] R. L. Freeman. *"Fundamentals of Telecommunications"*. John Wiley & Sons, Inc., page (57), Nov. 2005.

[30] L. Kleinrock. *"Theory: Queueing Systems"*. Wiley-Interscience, 1:(103), Nov. 1975.

[31] J. D. C. Little and S. C. Graves. *"Little's Law"*. D. Chhajed, T.J. Lowe (eds.) Building Institution: Insights from Basic Operations Management Models and Principles, 115:(81–100), 2008.

[32] Tenable. *"Set up a NAT Gateway"*. [online] Available at: https://docs.tenable.com/pvs/deployment/Content/GoogleCloudInstructionsNatGateway.htm.

Appendices

Appendix A

Test-Simulation Scripts

A.1 naive-uniform-site-migration.py

```
# M/M/c Queue is a Queue with only `c` servers and an infinite
    buffer.
# Basic Parameter of M/M/c queue:
#    1. Packet request Rate: `request`, the parameter of
   Possion R.V.
#    2. Packet Serving Rate: `service`, the parameter of Expo R
   .V.
#    3. Number of Servers:   `server`
#    4. sum:                 p0 + p1 + p2 + ... pc - pc

# Script Simulation: Uniform (Naive)

import matplotlib.pyplot as plt
import os
import math
import random
import math
import time
alpha=1.0
class Queue(object):
    def __init__(self, request, service, server):
        self.request = float(request)
        self.service = float(service)
        self.server = server
        self.utilization = request / (service * server)
        power = 1.0
        factor = 1.0
        sum = 1.0
        for i in range(1, server + 1):
            power *= request / self.service
            factor /= i
            sum += power * factor
        sum -= power * factor
        self.prob_sum = sum * (1.0 / (sum + ((power * factor)
           / (1 - self.utilization))))

    def avg_response_time(self):
        """
        1. Probability when a packet comes, it needs to queue in the buffer.
        That is, P(W>0) = 1 - P(N < c), Also known as Erlang-C function => Prob. Queue

        2. Average time of packets spending in the queue) => Average Queue Time

        3. Return the average time of packets spending in the system (in service and in the queue).
        i.e. (Prob. Queue / Average Queue Time ) + (1.0 / service)
        """
        return ((1.0 - self.prob_sum) / (self.server * self.service - self.request)) + 1.0 / self.service

def function(servers,request_id):
    request = request_id
    service = 15
    server = servers
    queue = Queue(request, service, server)
    avg_response_time = queue.avg_response_time()
    return avg_response_time

def iteration(num,n_zone):
    zone_1=[]
    zone_2=[]
    zone_3=[]
    zones={}
    resp_times={}
    n_vm_per_zone = 10
    n_vm = n_zone * n_vm_per_zone
    for zone in range(0,n_zone):
        zones['zone_'+str(zone+1)] = 0
        resp_times['resp_time_zone_'+str(zone+1)] = 0
    for index, key in enumerate(zones):
        zones[key] = int(n_vm / n_zone)
    for iteration in range(n_iterations):
        if iteration % 4000 == 0:
            num += 1
            if num > n_zone - 1:
                num = 0
            pass
        for index, key in enumerate(zones):
            resp_times['resp_time_'+key] = function(zones[key],request_id[key][num])
        average = sum(resp_times.values()) / len(resp_times)
        for key, val in resp_times.items():
            r = random.random()
            if resp_times[key] < average:
                prob = alpha * abs((average - resp_times[key]) / average)
                _a = resp_times[key]
                _temp = []
                for a, b in resp_times.items():
                    _temp.append(a)
                choice = random.choice(_temp)
                _get = choice.replace("resp_time_","")
```

```python
                move = key.replace("resp_time_","")
                get = _get
                if r < prob:
                    print(move,'to',get)
                    for index, key in enumerate(zones):
                        if zones[move] > 1:
                            zones[move] = zones[move] - 1
                            zones[get] = zones[get] + 1
                        break

        if iteration % 10 == 0:
            os.system('clear')
            for key, val in sorted(zones.items()):
                print ('VMs in',key, ':', val)
            for key, val in sorted(resp_times.items()):
                print (key,':',val)
                x = key.replace("resp_time_","")
                if x == 'zone_1':
                    zone_1.append(val)
                if x == 'zone_2':
                    zone_2.append(val)
                if x == 'zone_3':
                    zone_3.append(val)
            print('Iteration:', iteration, '/', n_iterations)
            print('-----------------------\n')
            pass
    return zone_1, zone_2, zone_3

def zone_avg(num,times,n_zone,zone_avg_1,zone_avg_2,zone_avg_3):
    for i in range(1,times+1):
        zone_avg_1[str(i)], zone_avg_2[str(i)], zone_avg_3[str(i)] = iteration(num,n_zone)
    return zone_avg_1, zone_avg_2, zone_avg_3
n_iterations=12000
n_zone=3
zone_1=[]
zone_2=[]
zone_3=[]
num = 0
request_id = {'zone_1': [60,70,35], 'zone_2': [70,35,60], 'zone_3': [35,60,70]}
zone_avg_1={}
zone_avg_2={}
zone_avg_3={}
times=100
x,y,z=zone_avg(num,times,n_zone,zone_avg_1,zone_avg_2,zone_avg_3)
_x=[sum(t)/times for t in zip(*x.values())]
_y=[sum(t)/times for t in zip(*y.values())]
_z=[sum(t)/times for t in zip(*z.values())]
plt.figure()
plt.title('Test-Simulation: Response Times - Uniform-Site Migration (Naive)')
plt.plot(_x, label='Zone 1', linestyle='--', linewidth=1.2)
plt.plot(_y, label='Zone 2', linestyle='--', linewidth=1.2)
plt.plot(_z, label='Zone 3', linestyle='--', linewidth=1.2)
plt.legend()
plt.xlabel('Number of Iterations')
plt.ylabel('Average Response Times (in secs)')
plt.xlim((0,(n_iterations/10)))
plt.tight_layout()
plt.savefig('./plots/plot_uniform_site_naive.png')
plt.close()
```

A.2 informed-uniform-site-migration.py

```
# M/M/c Queue is a Queue with only `c` servers and an infinite buffer.
# Basic Parameter of M/M/c queue:
#   1. Packet request Rate: `request`, the parameter of Possion R.V.
#   2. Packet Serving Rate: `service`, the parameter of Expo R.V.
#   3. Number of Servers:   `server`
#   4. sum:                 p0 + p1 + p2 + ... pc - pc

# Script Simulation: Uniform (Informed)

import matplotlib.pyplot as plt
import os
import math
import random
import math
import time
alpha=1.0
class Queue(object):
    def __init__(self, request, service, server):
        self.request = float(request)
        self.service = float(service)
        self.server = server
        self.utilization = request / (service * server)
        power = 1.0
        factor = 1.0
        sum = 1.0
        for i in range(1, server + 1):
            power *= request / self.service
            factor /= i
            sum += power * factor
        sum -= power * factor
        self.prob_sum = sum * (1.0 / (sum + ((power * factor)
            / (1 - self.utilization))))

    def avg_response_time(self):
        """
        1. Probability when a packet comes, it needs to queue
            in the buffer.
        That is, P(W>0) = 1 - P(N < c), Also known as Erlang-C
            function => Prob. Queue

        2. Average time of packets spending in the queue) =>
            Average Queue Time

        3. Return the average time of packets spending in the
           system (in service and in the queue).
        i.e. (Prob. Queue / Average Queue Time ) + (1.0 /
           service)
        """
        return ((1.0 - self.prob_sum) / (self.server * self.
           service - self.request)) + 1.0 / self.service

def function(servers,request_id):
    request = request_id
    service = 15
    server = servers
    queue = Queue(request, service, server)
    avg_response_time = queue.avg_response_time()
    return avg_response_time

def iteration(num,n_zone):
    zone_1=[]
    zone_2=[]
    zone_3=[]
    zones={}
    resp_times={}
    n_vm_per_zone = 10
    n_vm = n_zone * n_vm_per_zone
    for zone in range(0,n_zone):
        zones['zone_'+str(zone+1)] = 0
        resp_times['resp_time_zone_'+str(zone+1)] = 0
    for index, key in enumerate(zones):
        zones[key] = int(n_vm / n_zone)
    for iteration in range(n_iterations):
        if iteration % 4000 == 0:
            num += 1
            if num > n_zone - 1:
                num = 0
            pass
        for index, key in enumerate(zones):
            resp_times['resp_time_'+key] = function(zones[key
               ],request_id[key][num])
        average = sum(resp_times.values()) / len(resp_times)
        for key, val in resp_times.items():
            r = random.random()
            if resp_times[key] < average:
                prob = alpha * abs((average - resp_times[key])
                    / average)
                _a = resp_times[key]
                _temp = []
                for a, b in resp_times.items():
                    if b > _a:
                        _temp.append(a)
                _get=""
                if len(_temp) == 1:
                    _get = _temp[0].replace("resp_time_","")
                elif len(_temp) == 2:
                    choice = random.choice(_temp)
                    _get = choice.replace("resp_time_","")
                move = key.replace("resp_time_","")
                get = _get
                if r < prob:
                    print(move,'to',get)
                    for index, key in enumerate(zones):
                        if zones[move] > 1:
                            zones[move] = zones[move] - 1
                            zones[get] = zones[get] + 1
                        break

        if iteration % 10 == 0:
            os.system('clear')
            for key, val in sorted(zones.items()):
                print ('VMs in',key, ':', val)
            for key, val in sorted(resp_times.items()):
                print (key,':',val)
                x = key.replace("resp_time_","")
                if x == 'zone_1':
                    zone_1.append(val)
                if x == 'zone_2':
                    zone_2.append(val)
                if x == 'zone_3':
                    zone_3.append(val)
            print('Iteration:', iteration, '/', n_iterations)
            print('-----------------------\n')
            pass
    return zone_1, zone_2, zone_3

def zone_avg(num,times,n_zone,zone_avg_1,zone_avg_2,zone_avg_3
    ):
    for i in range(1,times+1):
        zone_avg_1[str(i)], zone_avg_2[str(i)], zone_avg_3[str
        (i)] = iteration(num,n_zone)
    return zone_avg_1, zone_avg_2, zone_avg_3
n_iterations=12000
n_zone=3
zone_1=[]
zone_2=[]
zone_3=[]
num = 0
request_id = {'zone_1': [60,70,35], 'zone_2': [70,35,60], '
    zone_3': [35,60,70]}
zone_avg_1={}
zone_avg_2={}
zone_avg_3={}
times=100
x,y,z=zone_avg(num,times,n_zone,zone_avg_1,zone_avg_2,
    zone_avg_3)
_x=[sum(t)/times for t in zip(*x.values())]
_y=[sum(t)/times for t in zip(*y.values())]
_z=[sum(t)/times for t in zip(*z.values())]
plt.figure()
plt.title('Test-Simulation: Response Times - Uniform-Site
    Migration (Informed)')
plt.plot(_x, label='Zone 1', linestyle='--', linewidth=1.2)
plt.plot(_y, label='Zone 2', linestyle='--', linewidth=1.2)
plt.plot(_z, label='Zone 3', linestyle='--', linewidth=1.2)
plt.legend()
plt.xlabel('Number of Iterations')
plt.ylabel('Average Response Times (in secs)')
plt.xlim((0,(n_iterations/10)))
plt.tight_layout()
plt.savefig('./plots/plot_uniform_site_informed.png')
plt.close()
```

A.3 biased-migration.py

```
# M/M/c Queue is a Queue with only ‘c‘ servers and an infinite
    buffer.
# Basic Parameter of M/M/c queue:
#   1. Packet request Rate: ‘request‘, the parameter of
    Possion R.V.
#   2. Packet Serving Rate: ‘service‘, the parameter of Expo R
    .V.
#   3. Number of Servers:   ‘server‘
#   4. sum:                 p0 + p1 + p2 + ... pc - pc

# Script Simulation: Biased

import matplotlib.pyplot as plt
import os
import math
import random
import math
import time
alpha=1.0
class Queue(object):
    def __init__(self, request, service, server):
        self.request = float(request)
        self.service = float(service)
        self.server = server
        self.utilization = request / (service * server)
        power = 1.0
        factor = 1.0
        sum = 1.0
        for i in range(1, server + 1):
            power *= request / self.service
            factor /= i
            sum += power * factor
        sum -= power * factor
        self.prob_sum = sum * (1.0 / (sum + ((power * factor)
           / (1 - self.utilization))))

    def avg_response_time(self):
        """
        1. Probability when a packet comes, it needs to queue
           in the buffer.
        That is, P(W>0) = 1 - P(N < c), Also known as Erlang-C
            function => Prob. Queue

        2. Average time of packets spending in the queue) =>
           Average Queue Time

        3. Return the average time of packets spending in the
           system (in service and in the queue).
        i.e. (Prob. Queue / Average Queue Time ) + (1.0 /
           service)
```

```python
        """
        return ((1.0 - self.prob_sum) / (self.server * self.service - self.request)) + 1.0 / self.service

def function(servers,request_id):
    request = request_id
    service = 15
    server = servers
    queue = Queue(request, service, server)
    avg_response_time = queue.avg_response_time()
    return avg_response_time

def iteration(num,n_zone):
    zone_1=[]
    zone_2=[]
    zone_3=[]
    zones={}
    resp_times={}
    n_vm_per_zone = 10
    n_vm = n_zone * n_vm_per_zone
    for zone in range(0,n_zone):
        zones['zone_'+str(zone+1)] = 0
        resp_times['resp_time_zone_'+str(zone+1)] = 0
    for index, key in enumerate(zones):
        zones[key] = int(n_vm / n_zone)
    for iteration in range(n_iterations):
        if iteration % 4000 == 0:
            num += 1
            if num > n_zone - 1:
                num = 0
            pass
        for index, key in enumerate(zones):
            resp_times['resp_time_'+key] = function(zones[key],request_id[key][num])
        average = sum(resp_times.values()) / len(resp_times)
        for key, val in resp_times.items():
            r = random.random()
            if resp_times[key] < average:
                prob_own =  abs((average - resp_times[key]) / average)
                _a = str(key)
                _Dict={}
                for key, val in sorted(resp_times.items()):
                    if str(_a) not in key:
                        _Dict[key]= val
                        _Dict_other={}
                        for key, val in _Dict.items():
                            _Dict_other["prob_"+key] = alpha * abs((average - resp_times[key]) / average)
                move = _a.replace("resp_time_","")
                get = max(_Dict_other, key=_Dict_other.get).replace("prob_resp_time_", "")
                if r < prob_own:
                    if zones[move] > 1:
                        zones[move] = zones[move] - 1
                        zones[get] = zones[get] + 1

        if iteration % 10 == 0:
            os.system('clear')
            for key, val in sorted(zones.items()):
                print ('VMs in',key, ':', val)
            for key, val in sorted(resp_times.items()):
                print (key,':',val)
                x = key.replace("resp_time_","")
                if x == 'zone_1':
                    zone_1.append(val)
                if x == 'zone_2':
                    zone_2.append(val)
                if x == 'zone_3':
                    zone_3.append(val)
            print('Iteration:', iteration, '/', n_iterations)
            print('----------------------\n')
            pass
    return zone_1, zone_2, zone_3

def zone_avg(num,times,n_zone,zone_avg_1,zone_avg_2,zone_avg_3
    ):
    for i in range(1,times+1):
        zone_avg_1[str(i)], zone_avg_2[str(i)], zone_avg_3[str
            (i)] = iteration(num,n_zone)
    return zone_avg_1, zone_avg_2, zone_avg_3

n_iterations=12000
n_zone=3
zone_1=[]
zone_2=[]
zone_3=[]
num = 0
request_id = {'zone_1': [60,70,35], 'zone_2': [70,35,60], '
    zone_3': [35,60,70]}
zone_avg_1={}
zone_avg_2={}
zone_avg_3={}
times=100
x,y,z=zone_avg(num,times,n_zone,zone_avg_1,zone_avg_2,
    zone_avg_3)
_x=[sum(t)/times for t in zip(*x.values())]
_y=[sum(t)/times for t in zip(*y.values())]
_z=[sum(t)/times for t in zip(*z.values())]
plt.figure()
plt.title('Test-Simulation: Response Times - Biased Migration'
    )
plt.plot(_x, label='Zone 1', linestyle='--', linewidth=1.2)
plt.plot(_y, label='Zone 2', linestyle='--', linewidth=1.2)
plt.plot(_z, label='Zone 3', linestyle='--', linewidth=1.2)
plt.legend()
plt.xlabel('Number of Iterations')
plt.ylabel('Average Response Times (in secs)')
plt.xlim((0,(n_iterations/10)))
plt.tight_layout()
plt.savefig('./plots/plot_biased.png')
plt.close()
```

A.4 single-point-migration.py

```
# M/M/c Queue is a Queue with only ‘c‘ servers and an infinite
    buffer.
# Basic Parameter of M/M/c queue:
#    1. Packet request Rate: ‘request‘, the parameter of
   Possion R.V.
#    2. Packet Serving Rate: ‘service‘, the parameter of Expo R
   .V.
#    3. Number of Servers:    ‘server‘
#    4. sum:                  p0 + p1 + p2 + ... pc - pc

# Script Simulation: Single Point

import matplotlib.pyplot as plt
import os
import math
import random
import math
import time
alpha = 1.0
class Queue(object):
    def __init__(self, request, service, server):
        self.request = float(request)
        self.service = float(service)
        self.server = server
        self.utilization = request / (service * server)
        power = 1.0
        factor = 1.0
        sum = 1.0
        for i in range(1, server + 1):
            power *= request / self.service
            factor /= i
            sum += power * factor
        sum -= power * factor
        self.prob_sum = sum * (1.0 / (sum + ((power * factor)
           / (1 - self.utilization))))

    def avg_response_time(self):
        """
        1. Probability when a packet comes, it needs to queue
           in the buffer.
        That is, P(W>0) = 1 - P(N < c), Also known as Erlang-C
            function => Prob. Queue

        2. Average time of packets spending in the queue) =>
           Average Queue Time

        3. Return the average time of packets spending in the
           system (in service and in the queue).
        i.e. (Prob. Queue / Average Queue Time ) + (1.0 /
           service)
        """
        return ((1.0 - self.prob_sum) / (self.server * self.
           service - self.request)) + 1.0 / self.service

def function(servers,request_id):
    request = request_id
    service = 15
    server = servers
    queue = Queue(request, service, server)
    avg_response_time = queue.avg_response_time()
    return avg_response_time

def iteration(num,n_zone):
    zone_1=[]
    zone_2=[]
    zone_3=[]
    zones={}
    resp_times={}
    n_vm_per_zone = 10
    n_vm = n_zone * n_vm_per_zone
    for zone in range(0,n_zone):
        zones['zone_'+str(zone+1)] = 0
        resp_times['resp_time_zone_'+str(zone+1)] = 0
    for index, key in enumerate(zones):
        zones[key] = int(n_vm / n_zone)
    for iteration in range(n_iterations):
        if iteration % 6000 == 0:
            num += 1
            if num > n_zone - 1:
                num = 0
            pass
        for index, key in enumerate(zones):
            resp_times['resp_time_'+key] = function(zones[key
                ],request_id[key][num])
        average = sum(resp_times.values()) / len(resp_times)
        for key, val in resp_times.items():
            r = random.random()
            if resp_times[key] < average:
                _a = str(key)
                _Dict={}
                for key, val in sorted(resp_times.items()):
                    if str(_a) not in key:
                        _Dict[key]= val
                prob = alpha * abs((average - resp_times[key])
                    / average)
                move = _a.replace("resp_time_","")
                get = max(_Dict, key=_Dict.get).replace("
                   resp_time_", "")
                if r < prob:
                    if zones[move] > 1:
                        zones[move] = zones[move] - 1
                        zones[get] = zones[get] + 1

        if iteration % 10 == 0:
            os.system('clear')
            for key, val in sorted(zones.items()):
                print ('VMs in',key, ':', val)
            for key, val in sorted(resp_times.items()):
                print (key,':',val)
                x = key.replace("resp_time_","")
                if x == 'zone_1':
                    zone_1.append(val)
                if x == 'zone_2':
                    zone_2.append(val)
```

```python
                if x == 'zone_3':
                    zone_3.append(val)
            print('Iteration:', iteration, '/', n_iterations)
            print('----------------------\n')
            pass
    return zone_1, zone_2, zone_3

def zone_avg(num,times,n_zone,zone_avg_1,zone_avg_2,zone_avg_3):
    for i in range(1,times+1):
        zone_avg_1[str(i)], zone_avg_2[str(i)], zone_avg_3[str(i)] = iteration(num,n_zone)
    return zone_avg_1, zone_avg_2, zone_avg_3
n_iterations=18000
n_zone=3
zone_1=[]
zone_2=[]
zone_3=[]
num = 0
request_id = {'zone_1': [60,70,35], 'zone_2': [70,35,60], 'zone_3': [35,60,70]}
zone_avg_1={}
zone_avg_2={}
zone_avg_3={}
times=100
x,y,z=zone_avg(num,times,n_zone,zone_avg_1,zone_avg_2,zone_avg_3)
_x=[sum(t)/times for t in zip(*x.values())]
_y=[sum(t)/times for t in zip(*y.values())]
_z=[sum(t)/times for t in zip(*z.values())]
plt.figure()
plt.title('Test-Simulation: Response Times - Single-Point Migration')
plt.plot(_x, label='Zone 1', linestyle='--', linewidth=1.2)
plt.plot(_y, label='Zone 2', linestyle='--', linewidth=1.2)
plt.plot(_z, label='Zone 3', linestyle='--', linewidth=1.2)
plt.legend()
plt.xlabel('Number of Iterations')
plt.ylabel('Average Response Times (in secs)')
plt.xlim((0,(n_iterations/10)))
plt.tight_layout()
plt.savefig('./plots/plot_single_point.png')
plt.close()
```

A.5 peer-to-peer-connection.py

```python
# M/M/c Queue is a Queue with only `c` servers and an infinite buffer.
# Basic Parameter of M/M/c queue:
#   1. Packet request Rate: `request`, the parameter of Possion R.V.
#   2. Packet Serving Rate: `service`, the parameter of Expo R.V.
#   3. Number of Servers:   `server`
#   4. sum:                 p0 + p1 + p2 + ... pc - pc

# Script Simulation: Peer-to-Peer

import matplotlib.pyplot as plt
import os
import math
import random
import math
import time
alpha=1.0
class Queue(object):
    def __init__(self, request, service, server):
        self.request = float(request)
        self.service = float(service)
        self.server = server
        self.utilization = request / (service * server)
        power = 1.0
        factor = 1.0
        sum = 1.0
        for i in range(1, server + 1):
            power *= request / self.service
            factor /= i
            sum += power * factor
        sum -= power * factor
        self.prob_sum = sum * (1.0 / (sum + ((power * factor) / (1 - self.utilization))))

    def avg_response_time(self):
        """
        1. Probability when a packet comes, it needs to queue in the buffer.
        That is, P(W>0) = 1 - P(N < c), Also known as Erlang-C function => Prob. Queue

        2. Average time of packets spending in the queue) => Average Queue Time

        3. Return the average time of packets spending in the system (in service and in the queue).
        i.e. (Prob. Queue / Average Queue Time ) + (1.0 / service)
        """
        return ((1.0 - self.prob_sum) / (self.server * self.service - self.request)) + 1.0 / self.service

def function(servers,request_id):
    request = request_id
    service = 15
    server = servers
    queue = Queue(request, service, server)
    avg_response_time = queue.avg_response_time()
    return avg_response_time

_groups = {'group_1' : ['zone_1', 'zone_2','zone_3'], 'group_2' : ['zone_4', 'zone_5','zone_6'], 'group_3': ['zone_3', 'zone_4']}
zone_1=[]
zone_2=[]
zone_3=[]
```

```
zone_4=[]
zone_5=[]
zone_6=[]
zones={}
resp_times={}

def groups(num,group,zones,resp_times):
        _temp={}
        for i in _groups[group]:
            if str('resp_time_'+i) in resp_times.keys():
                _temp['resp_time_'+i] = function(zones[i],
                    request_id[i][num])
                resp_times['resp_time_'+i] = function(zones[i
                    ], request_id[i][num])
        average = sum(_temp.values()) / len(_temp)
        for key, val in _temp.items():
            r = random.random()
            if _temp[key] < average:
                prob = alpha * abs((average - _temp[key]) /
                    average)
                _a = _temp[key]
                _x_temp = []
                for a,b in _temp.items():
                    if b > _a:
                        _x_temp.append(a)
                move = key.replace("resp_time_", "")
                get = random.choice(_x_temp).replace("
                    resp_time_","")
                if r < prob:
                    for index, key in enumerate(zones):
                        if zones[move] > 1:
                            zones[move] = zones[move] - 1
                            zones[get] = zones[get] + 1
                        break

def iteration(num,n_zone):
    zone_1=[]
    zone_2=[]
    zone_3=[]
    zone_4=[]
    zone_5=[]
    zone_6=[]
    zones={}
    resp_times={}
    n_vm_per_zone = 10
    n_vm = n_zone * n_vm_per_zone
    for zone in range(0,n_zone):
        zones['zone_'+str(zone+1)] = 0
        resp_times['resp_time_zone_'+str(zone+1)] = 0
    for index, key in enumerate(zones):
        zones[key] = int(n_vm / n_zone)
    for iteration in range(n_iterations):
        if iteration % 60000 == 0:
            num += 1
            if num > n_zone - 1:
                num = 0
            pass
        if iteration % 7 == 0:
```

```python
            groups(num,'group_1',zones,resp_times)
        if iteration % 7 == 0:
            groups(num,'group_2',zones,resp_times)
        if iteration % 7 == 0:
            groups(num,'group_3',zones,resp_times)

        if iteration % 100 == 0:
            os.system('clear')
            for key, val in sorted(zones.items()):
                print ('VMs in',key, ':', val)
            for key, val in sorted(resp_times.items()):
                print (key,':',val)
                x = key.replace("resp_time_","")
                if x == 'zone_1':
                    zone_1.append(val)
                if x == 'zone_2':
                    zone_2.append(val)
                if x == 'zone_3':
                    zone_3.append(val)
                if x == 'zone_4':
                    zone_4.append(val)
                if x == 'zone_5':
                    zone_5.append(val)
                if x == 'zone_6':
                    zone_6.append(val)
            print('Request ID Set:',num)
            print('Iteration:', iteration, '/', n_iterations)
            print('----------------------\n')
            pass
    return zone_1, zone_2, zone_3, zone_4, zone_5, zone_6

def zone_avg(num,times,n_zone,zone_avg_1,zone_avg_2,zone_avg_3,zone_avg_4,zone_avg_5,zone_avg_6):
    for i in range(1,times+1):
        zone_avg_1[str(i)], zone_avg_2[str(i)], zone_avg_3[str(i)] , zone_avg_4[str(i)], zone_avg_5[str(i)], zone_avg_6[str(i)]= iteration(num,n_zone)
    return zone_avg_1, zone_avg_2, zone_avg_3, zone_avg_4, zone_avg_5, zone_avg_6
n_iterations=180000
n_zone=6
num = 0
request_id = {'zone_1': [60,70,30,35,40,50], 'zone_2': [70,30,35,40,50,60], 'zone_3': [30,35,40,50,60,70], 'zone_4': [35,40,50,60,70,30], 'zone_5': [40,50,60,70,30,35], 'zone_6': [50,60,70,30,35,40]}
zone_avg_1={}
zone_avg_2={}
zone_avg_3={}
zone_avg_4={}
zone_avg_5={}
zone_avg_6={}
times=100
x,y,z,u,v,w=zone_avg(num,times,n_zone,zone_avg_1,zone_avg_2,zone_avg_3,zone_avg_4,zone_avg_5,zone_avg_6)
_x=[sum(t)/times for t in zip(*x.values())]
_y=[sum(t)/times for t in zip(*y.values())]
_z=[sum(t)/times for t in zip(*z.values())]
```

```
_u=[sum(t)/times for t in zip(*u.values())]
_v=[sum(t)/times for t in zip(*v.values())]
_w=[sum(t)/times for t in zip(*w.values())]
plt.figure()
plt.title('Test-Simulation: Response Times - Using Peer-to-
    Peer Connections')
plt.plot(_x, label='Zone 1', linestyle='--', linewidth=1.2)
plt.plot(_y, label='Zone 2', linestyle='--', linewidth=1.2)
plt.plot(_z, label='Zone 3', linestyle='--', linewidth=1.2)
plt.plot(_u, label='Zone 4', linestyle='--', linewidth=1.2)
plt.plot(_v, label='Zone 5', linestyle='--', linewidth=1.2)
plt.plot(_w, label='Zone 6', linestyle='--', linewidth=1.2)
plt.legend()
plt.xlabel('Number of Iterations')
plt.ylabel('Average Response Times (in secs)')
plt.xlim((0,(n_iterations/100)))
plt.tight_layout()
plt.savefig('./plots/plot_p_2_p.png')
plt.close()
```

A.6 uniform-graph-partition.py

```
# M/M/c Queue is a Queue with only `c` servers and an infinite
    buffer.
# Basic Parameter of M/M/c queue:
#   1. Packet request Rate: `request`, the parameter of
   Possion R.V.
#   2. Packet Serving Rate: `service`, the parameter of Expo R
   .V.
#   3. Number of Servers:   `server`
#   4. sum:                 p0 + p1 + p2 + ... pc - pc

# Script Simulation: Graph Partition (Uniform Informed)

import matplotlib.pyplot as plt
import os
import math
import random
import math
import time
alpha=1.0
class Queue(object):
    def __init__(self, request, service, server):
        self.request = float(request)
        self.service = float(service)
        self.server = server
        self.utilization = request / (service * server)
        power = 1.0
        factor = 1.0
        sum = 1.0
        for i in range(1, server + 1):
            power *= request / self.service
            factor /= i
            sum += power * factor
        sum -= power * factor
```

```python
        self.prob_sum = sum * (1.0 / (sum + ((power * factor) / (1 - self.utilization))))

    def avg_response_time(self):
        """
        1. Probability when a packet comes, it needs to queue in the buffer.
        That is, P(W>0) = 1 - P(N < c), Also known as Erlang-C function => Prob. Queue

        2. Average time of packets spending in the queue) => Average Queue Time

        3. Return the average time of packets spending in the system (in service and in the queue).
        i.e. (Prob. Queue / Average Queue Time ) + (1.0 / service)
        """
        return ((1.0 - self.prob_sum) / (self.server * self.service - self.request)) + 1.0 / self.service

def function(servers,request_id):
    request = request_id
    service = 15
    server = servers
    queue = Queue(request, service, server)
    avg_response_time = queue.avg_response_time()
    return avg_response_time

_groups = {'group_1' : ['zone_1', 'zone_2'], 'group_2' : ['zone_2', 'zone_3'], 'group_3': ['zone_3', 'zone_4'], 'group_4' : ['zone_4', 'zone_1']}
zone_1=[]
zone_2=[]
zone_3=[]
zone_4=[]
zones={}
resp_times={}

def groups(num,group,zones,resp_times):
        _temp={}
        for i in _groups[group]:
            if str('resp_time_'+i) in resp_times.keys():
                _temp['resp_time_'+i] = function(zones[i], request_id[i][num])
                resp_times['resp_time_'+i] = function(zones[i], request_id[i][num])
        average = sum(_temp.values()) / len(_temp)
        _a = _temp[min(_temp, key=_temp.get)]
        r = random.random()
        prob = alpha * abs((average - _a) / average)
        _min = _temp[min(_temp, key=_temp.get)]
        _max = _temp[max(_temp, key=_temp.get)]
        move = min(_temp, key=_temp.get).replace("resp_time_", "")
        get = max(_temp, key=_temp.get).replace("resp_time_", "")
        return average, _a, r, prob, _max, move, get,_temp
```

```

def _x_groups(num,group_i, group_j,zones,resp_times):
    average_gi, _a_gi, r_gi, prob_gi, _max_gi, move_gi, get_gi
       , _temp_gi = groups(num,group_i,zones,resp_times)
    average_gj, _a_gj, r_gj, prob_gj, _max_gj, move_gj, get_gj
       , _temp_gj = groups(num,group_j,zones,resp_times)
    _min_gi = min(_temp_gi, key=_temp_gi.get).replace("
       resp_time_","")
    _min_gj = min(_temp_gj, key=_temp_gj.get).replace("
       resp_time_","")
    _x_common = ''.join(list(set(group_i) & set(group_j)))
    if _min_gi == _min_gj:
        _max_g_i_j = max(max(_temp_gi, key=_temp_gi.get), max(
           _temp_gj, key=_temp_gj.get)).replace("resp_time_","
           ")
        for i in group_i:
            if i is not _x_common:
                _x_other_i = i
        for j in group_j:
            if j is not _x_common:
                _x_other_j = j
        if _max_g_i_j == _x_other_i:
            if _a_gi < average_gi:
                if r_gi < prob_gi:
                    if zones[_min_gi] > 1:
                        zones[_min_gi]=zones[_min_gi] - 1
                        zones[_max_g_i_j]=zones[_max_g_i_j] +
                            1
        if _max_g_i_j == _x_other_j:
            if _a_gj < average_gj:
                if r_gj < prob_gj:
                    if zones[_min_gj] > 1:
                        zones[_min_gj]=zones[_min_gj] - 1
                        zones[_max_g_i_j] =zones[_max_g_i_j] +
                            1
    else:
        if _a_gi < average_gi:
            if r_gi < prob_gi:
                if zones[move_gi] > 1:
                    zones[move_gi] = zones[move_gi] - 1
                    zones[get_gi] = zones[get_gi] + 1
        if _a_gj < average_gj:
            if r_gj < prob_gj:
                if zones[move_gj] > 1:
                    zones[move_gj] = zones[move_gj] - 1
                    zones[get_gj] = zones[get_gj]  + 1

def iteration(num,n_zone):
    zone_1=[]
    zone_2=[]
    zone_3=[]
    zone_4=[]
    zones={}
    resp_times={}
    n_vm_per_zone = 10
    n_vm = n_zone * n_vm_per_zone
    for zone in range(0,n_zone):
        zones['zone_'+str(zone+1)] = 0
        resp_times['resp_time_zone_'+str(zone+1)] = 0
    for index, key in enumerate(zones):
        zones[key] = int(n_vm / n_zone)
    for iteration in range(n_iterations):
        if iteration % 40000 == 0:
            num += 1
            if num > n_zone - 1:
                num = 0
            pass
        if iteration % 7 == 0:
            _x_groups(num,'group_1','group_2',zones,resp_times)
        if iteration % 7 == 0:
            _x_groups(num,'group_2','group_3',zones,resp_times)
        if iteration % 7 == 0:
            _x_groups(num,'group_3','group_4',zones,resp_times)
        if iteration % 7 == 0:
            _x_groups(num,'group_4','group_1',zones,resp_times)

        if iteration % 100 == 0:
            os.system('clear')
            for key, val in sorted(zones.items()):
                print ('VMs in',key, ':', val)
            for key, val in sorted(resp_times.items()):
                print (key,':',val)
                x = key.replace("resp_time_","")
                if x == 'zone_1':
                    zone_1.append(val)
                if x == 'zone_2':
                    zone_2.append(val)
                if x == 'zone_3':
                    zone_3.append(val)
                if x == 'zone_4':
                    zone_4.append(val)
            print('Request ID Set:',num)
            print('Iteration:', iteration, '/', n_iterations)
            print('----------------------\n')
            pass
    return zone_1, zone_2, zone_3, zone_4

def zone_avg(num,times,n_zone,zone_avg_1,zone_avg_2,zone_avg_3,zone_avg_4):
    for i in range(1,times+1):
        zone_avg_1[str(i)], zone_avg_2[str(i)], zone_avg_3[str(i)] , zone_avg_4[str(i)]= iteration(num,n_zone)
    return zone_avg_1, zone_avg_2, zone_avg_3, zone_avg_4
n_iterations=120000
n_zone=4
num = 0
request_id = {'zone_1': [60,70,30,35], 'zone_2': [70,30,35,60], 'zone_3': [30,35,60,70], 'zone_4': [35,60,70,30]}
zone_avg_1={}
zone_avg_2={}
zone_avg_3={}
```

```
zone_avg_4={}
times=100
x,y,z,u=zone_avg(num,times,n_zone,zone_avg_1,zone_avg_2,
    zone_avg_3,zone_avg_4)
_x=[sum(t)/times for t in zip(*x.values())]
_y=[sum(t)/times for t in zip(*y.values())]
_z=[sum(t)/times for t in zip(*z.values())]
_u=[sum(t)/times for t in zip(*u.values())]
plt.figure()
plt.title('Test-Simulation: Response Times - Using Graph
    Partitions')
plt.plot(_x, label='Zone 1', linestyle='--', linewidth=1.2)
plt.plot(_y, label='Zone 2', linestyle='--', linewidth=1.2)
plt.plot(_z, label='Zone 3', linestyle='--', linewidth=1.2)
plt.plot(_u, label='Zone 4', linestyle='--', linewidth=1.2)
plt.legend()
plt.xlabel('Number of Iterations')
plt.ylabel('Average Response Times (in secs)')
plt.xlim((0,(n_iterations/100)))
plt.tight_layout()
plt.savefig('./plots/plot_graph_partition.png')
plt.close()
```

Appendix B

Test-Bed Assisting Scripts

B.1 create-instance-group.sh

```bash
#!/bin/bash

# Copyright: Freeware (any user can download or distribute
   this script)
# Run the code at your own risk. Follow the instructions in
   README.
# Author: Samiul Saki
# Email: samiulsaki@gmail.com

# Script: Create Instance-Group

function progress {
    echo "$cmd"
        eval "$cmd" > /dev/null 2>&1 & PID=$! #simulate a long
           process
        printf "PROGRESS: ["
        # While process is running...
        while kill -0 $PID 2> /dev/null ; do
            printf  ">>"
            sleep 1
        done
        printf "] done!\n\n"
}

# Gets all the initial setup values
read -e -p "Enter loadbalancer name [ENTER for recommended]: "
    -i "lb-dc" lb
read -e -p "Enter address name [ENTER for recommended]: " -i "
   lb-ip-cr" address_name
read -e -p "Enter backend-service name [ENTER for recommended
   ]: " -i "web-map-backend-service" web_map
read -e -p "Enter url-map name [ENTER for recommended]: " -i "
   web-map" url_map
read -e -p "Enter target-http-proxy name [ENTER for
   recommended]: " -i "http-lb-proxy" target_proxy
read -e -p "Enter forwarding-rules name [ENTER for recommended
   ]: " -i "http-cr-rule" forward_rule

```

```bash
zones=$(gcloud compute instances list | tail -n+2 | grep $lb | grep -v TERMINATED | awk '{print $2}') # Finds all the data-centre region names

printf "\nSetting up Cross-Region:\n\n"

cmd="gcloud -q compute addresses create $address_name --ip-version=IPV4 --global" # Creates a address in gloud with the given name
progress;

eval "gcloud -q compute addresses list"

for i in $zones; do
    cmd="gcloud -q compute instance-groups unmanaged create $i-resources --zone $i" # Creates a instance-group for each region
    progress;
    lb_input=$(gcloud -q compute instances list | grep $i | grep $lb | awk '{print $1}')
    cmd="gcloud -q compute instance-groups unmanaged add-instances $i-resources --instances $lb_input --zone $i" # Adds the load-balancer instance of the same region n unmanaged instance-group
    progress;
done

eval "gcloud -q compute instance-groups list"

cmd="gcloud -q compute health-checks create http http-basic-check" # Creates a health check attribute
progress;

eval "gcloud -q compute health-checks list"

for i in $zones; do
    cmd="gcloud -q compute instance-groups unmanaged set-named-ports $i-resources --named-ports http:80 --zone $i" # For each instance group, define an HTTP service and map a port name to the relevant port
    progress;
done

cmd="gcloud -q compute backend-services create $web_map --protocol HTTP --health-checks http-basic-check --global" # Creates a backend service and specify its parameters. Set the --protocol field to HTTP because we are using HTTP to go to the instances
progress;

for i in $zones; do
    cmd="gcloud -q compute backend-services add-backend $web_map --balancing-mode UTILIZATION --max-utilization 0.5 --capacity-scaler 1 --instance-group $i-resources --instance-group-zone $i --global" # Adds instance-groups as backends to the backend- services. A backend defines the capacity (max CPU utilization or max queries per second) of the instance-groups it contains.
    progress;
done

eval "gcloud -q compute backend-services list"

cmd="gcloud -q compute url-maps create $url_map --default-
    service $web_map" # Creates a default URL map that directs
    all incoming requests to all instances
progress;

cmd="gcloud -q compute target-http-proxies create
    $target_proxy --url-map $url_map"
progress;

eval "gcloud -q compute target-http-proxies list"

lb_ip_address=$(gcloud compute addresses list | grep
    $address_name | awk '{print $2}')
cmd="gcloud -q compute forwarding-rules create $forward_rule
    --address $lb_ip_address --global --target-http-proxy
    $target_proxy --ports 80" # Creates a target HTTP proxy to
    route requests to your URL map
progress;

eval "gcloud -q compute forwarding-rules list"

printf "\nDone! Cross-Region Set.\n\n"

echo "THIS MIGHT TAKE SOME TIME TO PROPAGATE ${address_name
    ^^}. GIVE IT 10 MINUTES AND TRY AGAIN LATER."
```

B.2 dns.sh

```
#!/bin/bash

# Copyright: Freeware (any user can download or distribute
    this script)
# Run the code at your own risk. Follow the instructions in
    README.
# Author: Samiul Saki
# Email: samiulsaki@gmail.com

# Script: DNS Setup

# Gets inital setup values
lb="lb-dc"
project="project-brainiac"
echo "Setting Cloud DNS Records for zone '${project}'"
ALL_OLD_RECORDS=$(gcloud dns record-sets list --zone=$project
    | tail -n+2 | awk '{print $4}' | grep -oE "\b([0-9]{1,3}\.)
    {3}[0-9]{1,3}\b" | while read line; do echo "\"${line}\"";
    done | tr '\n' ' ') # Finds all the A record-sets of the
    given project name
ALL_NEW_RECORDS=$(gcloud compute forwarding-rules list | grep
    http-cr-rule | awk '{print $2}') # Finds the cross-region (
    CR) load-balancing IP address
```

```

eval "gcloud dns record-sets list --zone=$project"
echo "Old Records:"
eval "echo $ALL_OLD_RECORDS"
eval "gcloud dns record-sets transaction start --zone=$project"
eval "gcloud dns record-sets transaction remove --zone=$project --name="project-brainiac.eu." --type=A --ttl=300 $ALL_OLD_RECORDS" # Removes the old A record-sets of the project
echo "New Records:"
eval "echo $ALL_NEW_RECORDS"
eval "gcloud dns record-sets transaction add --zone=$project --name="project-brainiac.eu." --type=A --ttl=300 $ALL_NEW_RECORDS" # Adds the CR IP address to new A record-sets of the project
eval "gcloud dns record-sets transaction execute --zone=$project"
eval "gcloud dns record-sets list --zone=$project" # Sets the A record-sets of the project
echo "Done! Records Set"

echo "THIS MIGHT TAKE SOME TIME TO PROPAGATE ${project^^}.EU. GIVE IT SOME MINUTES AND TRY AGAIN LATER."
```

Appendix C

Test-Bed Startup Scripts

C.1 consul-master.sh

```
#! /bin/bash

# Script: Consul Startup

sudo su -
sudo apt-get update
sudo timedatectl set-timezone Europe/Oslo # Sets the timezone
sleep 5
apt-get install -y git unzip curl jq # Install packages
cd /tmp/

# Setup Consul
wget https://releases.hashicorp.com/consul/1.0.6/consul_1.0.6
    _linux_amd64.zip
unzip consul_1.0.6_linux_amd64.zip
mv consul /usr/local/bin
mkdir -p /var/lib/consul/
mkdir -p /usr/share/consul
mkdir -p /etc/consul.d
cd ~
consul_hostname=$(hostname)
consul_ip=$(hostname -I)
consul_dc=$(hostname | sed "s/consul-master-//g")

cat <<EOF > /etc/consul.d/ui.json
{
  "addresses": {
    "http": "0.0.0.0"
  }
}
EOF

# Configure instance as consul server in the cluster and add
    as the service
cat <<EOF > /etc/systemd/system/consul.service
[Unit]
Description=Consul
Documentation=https://www.consul.io/

```

```
[Service]
ExecStart=/usr/local/bin/consul agent -server -ui -bootstrap-
    expect=2 -data-dir=/tmp/consul -node=${consul_hostname} -
    bind=${consul_ip} -datacenter ${consul_dc} -config-dir=/etc
    /consul.d/
ExecReload=/bin/kill -HUP $MAINPID
LimitNOFILE=65536

[Install]
WantedBy=multi-user.target
EOF

# Consul service started and added to system startup
sudo systemctl daemon-reload && sudo systemctl start consul.
    service && sudo systemctl enable consul.service
consul members
```

C.2 lb.sh

```
#! /bin/bash

# Script: Load-Balancer Startup

sudo su -
sudo apt-get update
add-apt-repository ppa:vbernat/haproxy-1.7 -y
sudo timedatectl set-timezone Europe/Oslo # Sets the timezone
sleep 5
apt-get update
apt-get install -y haproxy git unzip jq # Install packages
cd /tmp/

# Setup Consul
wget https://releases.hashicorp.com/consul/1.0.6/consul_1.0.6
    _linux_amd64.zip
unzip consul_1.0.6_linux_amd64.zip
mv consul /usr/local/bin
mkdir -p /var/lib/consul/
mkdir -p /usr/share/consul
mkdir -p /etc/consul.d
cd ~

lb_hostname=$(hostname)
lb_ip=$(hostname -I)
lb_dc=$(hostname | sed "s/lb-//g")
consul_ip=$(gcloud compute instances list | tail -n+2 | grep
    consul-master-${lb_dc} | awk '{print $4}')

# Configure instance as consul server in the cluster and add
    as the service
cat <<EOF > /etc/systemd/system/consul.service
[Unit]
Description=Consul
Documentation=https://www.consul.io/

[Service]
```

```
ExecStart=/usr/local/bin/consul agent -server -join=${consul_ip} -data-dir=/tmp/consul -node=${lb_hostname} -bind=${lb_ip} -datacenter ${lb_dc} -config-dir=/etc/consul.d/
ExecReload=/bin/kill -HUP $MAINPID
LimitNOFILE=65536

[Install]
WantedBy=multi-user.target
EOF

# Consul service started and added to system startup
sudo systemctl daemon-reload && sudo systemctl start consul.service && sudo systemctl enable consul.service

cd /tmp/
# Setup Consul-HAProxy
wget https://github.com/hashicorp/consul-haproxy/releases/download/v0.2.0/consul-haproxy_0.2.0_linux_amd64.tar.gz
tar -xvzf consul-haproxy_0.2.0_linux_amd64.tar.gz
sudo chmod +x consul-haproxy_0.2.0_linux_amd64/consul-haproxy
mv consul-haproxy_0.2.0_linux_amd64/consul-haproxy /usr/local/bin/
cd ~

# Initiate a HAProxy template
cat <<EOF> /etc/consul.d/consul_ha.cfg
global
        log /dev/log    local0
        log /dev/log    local1 notice
        chroot /var/lib/haproxy
        stats socket /run/haproxy/admin.sock mode 660 level admin
        stats timeout 30s
        user haproxy
        group haproxy
        daemon
        # Default SSL material locations
        ca-base /etc/ssl/certs
        crt-base /etc/ssl/private
        # Default ciphers to use on SSL-enabled listening sockets.
        # For more information, see ciphers(1SSL). This list is from:
        #  https://hynek.me/articles/hardening-your-web-servers-ssl-ciphers/
        # An alternative list with additional directives can be obtained from
        #  https://mozilla.github.io/server-side-tls/ssl-config-generator/?server=haproxy
        ssl-default-bind-ciphers ECDH+AESGCM:DH+AESGCM:ECDH+AES256:DH+AES256:ECDH+AES128:DH+AES:RSA+AESGCM:RSA+AES:!aNULL:$
        ssl-default-bind-options no-sslv3
defaults
        log     global
        mode    http
        option  httplog
        option  dontlognull
        timeout connect 5000
        timeout client  50000
        timeout server  50000
        errorfile 400 /etc/haproxy/errors/400.http
        errorfile 403 /etc/haproxy/errors/403.http
        errorfile 408 /etc/haproxy/errors/408.http
        errorfile 500 /etc/haproxy/errors/500.http
        errorfile 502 /etc/haproxy/errors/502.http
        errorfile 503 /etc/haproxy/errors/503.http
        errorfile 504 /etc/haproxy/errors/504.http

frontend http_front
   bind *:80
   stats uri /haproxy?stats
   default_backend http_back

backend http_back
   balance roundrobin
   {{range .c}}
   {{.}}{{end}}

listen stats
   bind *:9000
   mode http
   stats enable
   stats show-node
   stats show-legends
   stats refresh 20s
   stats uri /
   stats realm HAproxy-Statistics
   stats hide-version
EOF

systemctl restart haproxy
# Start a background process for consul-haproxy to add the backend-servers dynamically when the a worker instance register/deregister
nohup sudo consul-haproxy -addr=localhost:8500 -in /etc/consul.d/consul_ha.cfg -backend "c=webserver@${lb_dc}:80" -out /etc/haproxy/haproxy.cfg -reload "/etc/init.d/haproxy restart" &>/dev/null & disown
echo 'lb_dc=$(hostname | sed "s/lb-//g") && nohup sudo consul-haproxy -addr=localhost:8500 -in /etc/consul.d/consul_ha.cfg -backend "c=webserver@${lb_dc}:80" -out /etc/haproxy/haproxy.cfg -reload "/etc/init.d/haproxy restart" &>/dev/null & disown && exit 0' | sudo tee /etc/rc.local # Set the above command in the system startup
```

C.3 nat-gateway.sh

```
#! /bin/bash

# Script: Nat-Gateway Startup

sudo su -
sudo apt-get update
sudo timedatectl set-timezone Europe/Oslo # Sets the timezone
sleep 5
# Enable ip forwarding for nat-gateway
sudo sh -c "echo 1 > /proc/sys/net/ipv4/ip_forward"
sudo iptables -t nat -A POSTROUTING -o ens4 -j MASQUERADE
```

C.4 worker.sh

```
#! /bin/bash

# Script: Worker Startup

sudo su -
sudo apt-get update
sudo timedatectl set-timezone Europe/Oslo # Sets the timezone
sleep 5
apt-get install -y git unzip bc jq nginx # Install packages
ufw allow 'Nginx HTTP'
systemctl start nginx # Starts Nginx

# Setup Consul
cd /tmp/
wget https://releases.hashicorp.com/consul/1.0.6/consul_1.0.6_linux_amd64.zip
unzip consul_1.0.6_linux_amd64.zip
mv consul /usr/local/bin
mkdir -p /var/lib/consul/
mkdir -p /usr/share/consul
mkdir -p /etc/consul.d
cd ~

echo '{"service": {"name": "webserver", "tags": ["HTTP"], "port": 80}}' | sudo tee /etc/consul.d/webserver.json # Creates a web server

worker_hostname=$(hostname)
worker_ip=$(hostname -I)
worker_dc=$(hostname | cut -c8-10)

nohup sudo consul agent -server=false -data-dir=/tmp/consul -node=${worker_hostname} -bind=${worker_ip} -enable-script-checks=true -datacenter ${worker_dc} -config-dir=/etc/consul.d &>/dev/null & sleep 3 && consul join consul-master-${worker_dc} # Registers to the consul-cluster as client
echo 'worker_hostname=$(hostname) && worker_ip=$(hostname -I) && worker_dc=$(hostname | cut -c8-10) && nohup sudo consul agent -server=false -data-dir=/tmp/consul -node=${worker_hostname} -bind=${worker_ip} -enable-script-checks=true -datacenter ${worker_dc} -config-dir=/etc/consul.d &>/dev/null & sleep 3 && consul join consul-master-${worker_dc} && exit 0' | sudo tee /etc/rc.local # Save the above command at system startup

cd /tmp/
git clone https://github.com/samiulsaki/project-brainiac.git
```

```
nohup sudo /bin/bash -c 'while [ true ]; do sleep 30; /tmp/project-brainiac/sidekicks/csv-gen.sh; done' &>/dev/null & disown # Starts CSV generator script in worker instance
nohup sudo /bin/bash -c 'while [ true ]; do timer=$(( ( RANDOM % 300 )  + 300 )); sleep $timer; /tmp/project-brainiac/startups/worker-[variant]/[variant].sh; done' &>/dev/null & disown # Starts the migration script in worker instance
```

Appendix D

Test-Bed Migration Scripts

D.1 csv-gen.sh

```bash
#!/bin/bash

# Copyright: Freeware (any user can download or distribute this script)
# Run the code at your own risk. Follow the instructions in README.
# Author: Samiul Saki
# Email: samiulsaki@gmail.com

# Script: CSV Generator

lb=$(gcloud compute instances list | grep lb-dc | grep -v TERMINATED | awk '{print $1}') # Finds all the load-balancers in the same network

for argh in $lb; do
        lb_ip=$(gcloud compute instances list | grep $argh | awk '{print $5}') # Find the IP address of a particular load-balancer
        rtime=$(eval "curl -sSL 'http://$lb_ip/haproxy?stats;csv;norefresh' | cut -d "," -f 2,61 | column -s, -t | grep worker | awk '{print \$2}' ") # Finds the number of back-end servers (workers) and the response time of each server of the load-balancer
        sum=0
        for i in $rtime; do
                sum=$((sum + i)) # Sum up all the workers total response time
        done
        echo "$(( sum / $(echo "${rtime}" | wc -l) ))" >> /home/ubuntu/$argh.csv # Calculates the average response time of the load-balancer and append to the CSV file
done
```

D.2 migrate-uniform-naive.sh

```bash
#! /bin/bash

# Copyright: Freeware (any user can download or distribute this script)
# Run the code at your own risk. Follow the instructions in README.
# Author: Samiul Saki
# Email: samiulsaki@gmail.com

# Script: Migration Uniform (Naive) [move to any datacentre]

# Gets initial values
serv_account="my-instances@samiulsaki-148219.iam.gserviceaccount.com"
own_hostname=$(hostname)
own_dc=$(hostname | cut -c8-10)
own_zone=$(gcloud compute instances list | grep $own_hostname | awk '{print $2}')

CSVS=$(ls /home/ubuntu/) # Reads through all the lb logs for alive data-centre (dc) created by csv-gen script
avg_resp=0

for argh in $CSVS; do
        resp=0
        for i in $(tail -n 50 /home/ubuntu/${argh}); do # Read the last 50 entries (average response time)
                resp=$(( resp + i ))
        done
        eval "var=$(echo $argh | sed "s/lb-//g" | sed "s/.csv//g")" # Create a variable with dc name
        eval "${var}=$resp" # Add the response time to corresponding dc varible
        avg_resp=$(( avg_resp + resp ))
done

avg_resp=$(( avg_resp / $(echo "${CSVS}" | wc -l) )) # Calculates average total response time of all the alive dc
own_resp=$((own_dc))
total_zones=$(gcloud compute instances list | grep lb-dc | grep -v TERMINATED | awk '{print $1}' ) # Finds all the alive dc name
cand_dc=$(echo $total_zones | xargs shuf -n1 -e | sed "s/lb-//g") # Choose a random dc as candidate zone
cand_zone=$(gcloud compute instances list | grep lb-$cand_dc | awk '{print $2}') # Finds the gcloud region of the candidate zone
cand_dc_max_inst=$(gcloud compute instances list | grep worker-$cand_dc | awk '{print $1}' | grep -o '[0-9]*$' | wc -l) # Finds the existing number of worker instances in dc
own_dc_alive_inst=$(gcloud compute instances list | grep worker-$own_dc |  grep -v TERMINATED | grep -v STOPPING | awk '{print $1}' | grep -o '[0-9]*$' | wc -l) # Finds the exsiting number of alive worker instances in own dc
random=$(( ( RANDOM % 1000 )  + 1 ))

```

```
migrate_thres=3 # Set in minimum threshold limit for how many VMs per zone
migrate_resp=1
temp_avg=$( echo "$avg_resp / 1000" | bc -l)
temp_limit=$( echo "$migrate_resp /1000" | bc -l ) # Set in threshold for total system average response time to migrate

# If own dc have more than 3 alive worker instances, if candidate dc is not same as own dc, if own response time is less than average total response time, if system total average response time is higher than own zone average response time
if [ $own_dc_alive_inst -gt $migrate_thres ] && [ $cand_dc != $own_dc ] && [ $own_resp -lt $avg_resp ] && (( $(echo "$temp_avg > $temp_limit" | bc -l) ))
then
    eval "gcloud -q compute instances create worker-$cand_dc-$((cand_dc_max_inst + 1))-$random --network default --no-address --image ubuntu-console --machine-type f1-micro --zone $cand_zone --tags no-ip-$cand_dc --service-account $serv_account --scopes cloud-platform --metadata-from-file startup-script=/tmp/project-brainiac/startups/worker-uniform-naive/worker-uniform-naive.sh" # Creates a new worker instance at candidate dc
    eval "consul leave -http-addr=127.0.0.1:8500" # Gracefully removes itself from consul cluster
    sleep 2
    eval "gcloud -q compute instances delete $own_hostname --zone $own_zone" # Delete itself from gcloud
fi
```

D.3 migrate-uniform-informed.sh

```
#! /bin/bash

# Copyright: Freeware (any user can download or distribute this script)
# Run the code at your own risk. Follow the instructions in README.
# Author: Samiul Saki
# Email: samiulsaki@gmail.com

# Script: Migration Unform (Informed) [move to the datacentre which is only worse (rondomly)]

# Gets initial values
serv_account="my-instances@samiulsaki-148219.iam.gserviceaccount.com"
own_hostname=$(hostname)
own_dc=$(hostname | cut -c8-10)
own_zone=$(gcloud compute instances list | grep $own_hostname | awk '{print $2}')

CSVS=$(ls /home/ubuntu/) # Reads through all the lb logs for alive data-centre (dc) created by csv-gen script
avg_resp=0
```

```

for argh in $CSVS; do
        resp=0
        for i in $(tail -n 50 /home/ubuntu/${argh}); do # Read the last 50 entries (average response time)
                resp=$(( resp + i ))
        done
        eval "var=$(echo $argh | sed "s/lb-//g" | sed "s/.csv//g")" # Create a variable with dc name
        eval "${var}=$resp" # Add the response time to corresponding dc varible
    avg_resp=$((avg_resp + resp ))
done

avg_resp=$(( avg_resp / $(echo "${CSVS}" | wc -l) )) # Calculates average total response time of all the alive dc
own_resp=$((own_dc))

cand_dc=$(for i in $(seq 1 $(echo "${CSVS}" | wc -l)); do name=dc$i; value=${!name}; echo "$name $value"; done | sort -k2n | grep -v $own_dc | awk -v x=$((own_resp)) '{ p=$1; q=$2; if (q > x) print ; }' | awk '{print $1}' | xargs shuf -n1 -e) # Choose a candidate dc (in random) which has the higher average response time
cand_zone=$(gcloud compute instances list | grep lb-$cand_dc | awk '{print $2}') # Finds the gcloud region of the candidate zone
cand_dc_max_inst=$(gcloud compute instances list | grep worker-$cand_dc | awk '{print $1}' | grep -o '[0-9]*$' | wc -l) # Finds the existing number of worker instances in dc
own_dc_alive_inst=$(gcloud compute instances list | grep worker-$own_dc |  grep -v TERMINATED | grep -v STOPPING | awk '{print $1}' | grep -o '[0-9]*$' | wc -l) # Finds the exsiting number of alive worker instances in own dc
random=$(( ( RANDOM % 1000 )  + 1 ))

migrate_thres=3 # Set in minimum threshold limit for how many VMs per zone
migrate_resp=1
temp_avg=$( echo "$avg_resp / 1000" | bc -l)
temp_limit=$( echo "$migrate_resp /1000" | bc -l ) # Set in threshold for total system average response time to migrate

# If own dc have more than 3 alive worker instances, if candidate dc is not same as own dc, if own response time is less than average total response time, if system total average response time is higher than own zone average response time
if [ $own_dc_alive_inst -gt $migrate_thres ] && [ $cand_dc != $own_dc ] && [ $own_resp -lt $avg_resp ] && (( $(echo "$temp_avg > $temp_limit" | bc -l) ))
then
    eval "gcloud -q compute instances create worker-$cand_dc-$((cand_dc_max_inst + 1))-$random --network default --no-address --image ubuntu-console --machine-type f1-micro --zone $cand_zone --tags no-ip-$cand_dc --service-account $serv_account --scopes cloud-platform --metadata-from-file startup-script=/tmp/project-brainiac
```

```
        /startups/worker-uniform-informed/worker-uniform-
        informed.sh" # Creates a new worker instance at
        candidate dc
    eval "consul leave -http-addr=127.0.0.1:8500" # Gracefully
         removes itself from consul cluster
    sleep 2
    eval "gcloud -q compute instances delete $own_hostname --
        zone $own_zone" # Delete itself from gcloud
fi
```

D.4 migrate-biased.sh

```
#! /bin/bash

# Copyright: Freeware (any user can download or distribute
    this script)
# Run the code at your own risk. Follow the instructions in
    README.
# Author: Samiul Saki
# Email: samiulsaki@gmail.com

# Script: Migration Biased [move to the worst datacentre with
    highest probability]

# Gets initial values
serv_account="my-instances@samiulsaki-148219.iam.
    gserviceaccount.com"
own_hostname=$(hostname)
own_dc=$(hostname | cut -c8-10)
own_zone=$(gcloud compute instances list | grep $own_hostname
    | awk '{print $2}')

CSVS=$(ls /home/ubuntu/) # Reads through all the lb logs for
    alive data-centre (dc) created by csv-gen script
avg_resp=0
declare -A array_temp # Temporary array for response time

for argh in $CSVS; do
    resp=0
    for i in $(tail -n 50 /home/ubuntu/${argh}); do # Read the
         last 50 entries (average response time)
         resp=$(( resp + i ))
    done
    eval "var=$(echo $argh | sed "s/lb-//g" | sed "s/.csv//g")
        " # Create a variable with dc name
    eval "${var}=$resp" # Add the response time to
        corresponding dc varible
    avg_resp=$((avg_resp + resp ))
    array_temp+=(["$var"]="${resp}") # Add the response time
        to temporary array with dc variable as index
done

avg_resp=$(( avg_resp / $(echo "${CSVS}" | wc -l) )) #
    Calculates average total response time of all the alive dc
own_resp=$((own_dc))
declare -A array_prob # Array for probability
```

```

for i in "${!array_temp[@]}"; do
    x=${array_temp[$i]}
    if [ $x -eq "0" ]
    then
        x=$((1))
    fi
    if [ $x != 0 ] && [ "$i" != "$own_dc" ] # Includes all the
        dc response time except own dc
    then
        c=$(echo "($avg_resp - $x) / $avg_resp" | bc -l) #
            Calculates the probability to move for each dc
        array_prob+=(["$i"]="${c}") # Add the probability to
            probability array for with dc variable as index
    fi
done

# Choose a dc probability as initial maximum probability
t_keys=(${!array_prob[@]})
t_size=${#array_prob[@]}
t_random_index=$(( RANDOM % t_size ))
cand_dc="${t_keys[${t_random_index}]}"
max=${array_prob[$cand_dc]}

for j in "${!array_prob[@]}"; do
    u=$(echo "${array_prob[$j]}" | bc -l)
    v=$(echo "$max" | bc -l)
    w=$(echo "$u < $v" | bc -l)
    if [ $w == '1' ]; then # Finds the probability that it
        closest to moving.
        max=$j
        cand_dc=$j # Choose the candidate zone with highest
            probability to move
    fi
done

cand_zone=$(gcloud compute instances list | grep lb-$cand_dc |
     awk '{print $2}') # Finds the gcloud region of the
    candidate zone
cand_dc_max_inst=$(gcloud compute instances list | grep worker
    -$cand_dc | awk '{print $1}' | grep -o '[0-9]*$' | wc -l) #
     Finds the existing number of worker instances in dc
own_dc_alive_inst=$(gcloud compute instances list | grep
    worker-$own_dc |  grep -v TERMINATED | grep -v STOPPING |
    awk '{print $1}' | grep -o '[0-9]*$' | wc -l) # Finds the
    exsiting number of alive worker instances in own dc
random=$(( ( RANDOM % 1000 )  + 1 ))

migrate_thres=3 # Set in minimum threshold limit for how many
    VMs per zone
migrate_resp=1
temp_avg=$( echo "$avg_resp / 1000" | bc -l)
temp_limit=$( echo "$migrate_resp /1000" | bc -l ) # Set in
    threshold for total system average response time to migrate

# If own dc have more than 3 alive worker instances, if
    candidate dc is not same as own dc, if own response time is
     less than average total response time, if system total
    average response time is higher than own zone average
    response time
if [ $own_dc_alive_inst -gt $migrate_thres ] && [ $cand_dc !=
    $own_dc ] && [ $own_resp -lt $avg_resp ] && (( $(echo "
    $temp_avg > $temp_limit" | bc -l) ))
then
    eval "gcloud -q compute instances create worker-$cand_dc-$
        ((cand_dc_max_inst + 1))-$random --network default --no
        -address --image ubuntu-console --machine-type f1-micro
         --zone $cand_zone --tags no-ip-$cand_dc --service-
        account $serv_account --scopes cloud-platform --
        metadata-from-file startup-script=/tmp/project-brainiac
        /startups/worker-biased/worker-biased.sh" # Creates a
        new worker instance at candidate dc
    eval "consul leave -http-addr=127.0.0.1:8500" # Creates a
        new worker instance at candidate dc
    sleep 2
    eval "gcloud -q compute instances delete $own_hostname --
        zone $own_zone" # Delete itself from gcloud
fi
```

D.5 migrate-single-point.sh

```
#! /bin/bash

# Copyright: Freeware (any user can download or distribute
    this script)
# Run the code at your own risk. Follow the instructions in
    README.
# Author: Samiul Saki
# Email: samiulsaki@gmail.com

# Script: Migration Single Point [move to the worst datacentre
    ]

# Gets initial values
serv_account="my-instances@samiulsaki-148219.iam.
    gserviceaccount.com"
own_hostname=$(hostname)
own_dc=$(hostname | cut -c8-10)
own_zone=$(gcloud compute instances list | grep $own_hostname
    | awk '{print $2}')

CSVS=$(ls /home/ubuntu/) # Reads through all the lb logs for
    alive data-centre (dc) created by csv-gen script
avg_resp=0
for argh in $CSVS; do
    resp=0
    for i in $(tail -n 100 /home/ubuntu/${argh}); do # Read
        the last 50 entries (average response time)
         resp=$(( resp + i ))
    done
    eval "var=$(echo $argh | sed "s/lb-//g" | sed "s/.csv//g")
        " # Create a variable with dc name
    eval "${var}=$resp" # Add the response time to
        corresponding dc varible
```

```bash
    avg_resp=$(( avg_resp + resp ))
done

avg_resp=$(( avg_resp / $(echo "${CSVS}" | wc -l) )) # Calculates average total response time of all the alive dc
own_resp=$(( own_dc ))
cand_dc=$(for i in $(seq 1 $(echo "${CSVS}" | wc -l)); do name=dc$i; value=${!name}; echo "$name $value"; done | grep -v $own_dc | sort -k2n | tail -1 | awk '{print $1}') # Choose a candidate dc which has the highest average response time
cand_zone=$(gcloud compute instances list | grep lb-$cand_dc | awk '{print $2}') # Finds the gcloud region of the candidate zone
cand_dc_max_inst=$(gcloud compute instances list | grep worker-$cand_dc | awk '{print $1}' | grep -o '[0-9]*$' | wc -l) # Finds the existing number of worker instances in dc
own_dc_alive_inst=$(gcloud compute instances list | grep worker-$own_dc |  grep -v TERMINATED | grep -v STOPPING | awk '{print $1}' | grep -o '[0-9]*$' | wc -l) # Finds the exsiting number of alive worker instances in own dc
random=$(( ( RANDOM % 1000 )  + 1 ))

migrate_thres=3 # Set in minimum threshold limit for how many VMs per zone
migrate_resp=1
temp_avg=$( echo "$avg_resp / 1000" | bc -l)
temp_limit=$( echo "$migrate_resp /1000" | bc -l ) # Set in threshold for total system average response time to migrate

# If own dc have more than 3 alive worker instances, if candidate dc is not same as own dc, if own response time is less than average total response time, if system total average response time is higher than own zone average response time
if [ $own_dc_alive_inst -gt $migrate_thres ] && [ $cand_dc != $own_dc ] && [ $own_resp -lt $avg_resp ] && (( $(echo "$temp_avg > $temp_limit" | bc -l) ))
then
    eval "gcloud -q compute instances create worker-$cand_dc-$((cand_dc_max_inst + 1))-$random --network default --no-address --image ubuntu-console --machine-type f1-micro --zone $cand_zone --tags no-ip-$cand_dc --service-account $serv_account --scopes cloud-platform --metadata-from-file startup-script=/tmp/project-brainiac/startups/worker-single-point/worker-single-point.sh" # Creates a new worker instance at candidate dc
    eval "consul leave -http-addr=127.0.0.1:8500" # Gracefully removes itself from consul cluster
    sleep 2
    eval "gcloud -q compute instances delete $own_hostname --zone $own_zone" # Delete itself from gcloud
fi
```

Appendix E

Test-Bed Initial Script

E.1 automate.sh

```bash
#!/bin/bash

# Copyright: Freeware (any user can download or distribute
    this script)
# Run the code at your own risk. Follow the instructions in
    README.
# Author: Samiul Saki
# Email: samiulsaki@gmail.com

# Script: Initial Setup

function progress {
    echo "$cmd"
    eval "$cmd" > /dev/null 2>&1 & PID=$! #simulate a long
        process
    printf "PROGRESS: ["
    # While process is running...
    while kill -0 $PID 2> /dev/null ; do
        printf  ">>"
        sleep 1
        done
        printf "] done!\n\n"
}

function instance_lists {
        eval "gcloud compute instances list"
}

clear

echo "This is the initial infrustructure setup for the
    autonomous VMs. Follow the instructions."

instance_lists

# Gets all the initial setup values
read -e -p "Enter zone names [separated by comma]: e.g. " -i "
    us-central1-a,southamerica-east1-b,europe-west3-c,asia-
    northeast1-c" zone_name
```

```
z=$(echo "$zone_name" | sed 's/,/ /g' | wc -w )
read -e -p "Enter how many zones [Should match the previous agrument] " -i "$z" zone_number # Regions are separated by zone-names separated by commas
read -e -p "Enter images to use [ENTER for recommended]: " -i "ubuntu-console" image
read -e -p "Enter loadbalancer name [ENTER for recommended]: " -i "lb-dc" lb
read -e -p "Enter loadbalancer machine type [ENTER for recommended]: " -i "n1-standard-2" lb_machine
read -e -p "Enter consul-master name [ENTER for recommended]: " -i "consul-master-dc" consul
read -e -p "Enter consul-master machine type [ENTER for recommended]: " -i "n1-standard-1" consul_machine
read -e -p "Enter instance name [ENTER for recommended]: " -i "worker" inst
read -e -p "Enter instance machine type [ENTER for recommended]: " -i "f1-micro" inst_machine
read -e -p "Enter how many instances per loadbalancer [ENTER for recommended]: " -i "30" inst_number
read -e -p "Enter the variants name [ENTER uniform-naive/uniform-informed/biased/single-point]: " -i "single-point" variant # Migration script to imply in the network
read -e -p "Enter the service account email [ENTER for recommended]: " -i "my-instances@samiulsaki-148219.iam.gserviceaccount.com" serv_account

max_lb=$(gcloud compute instances list | grep $lb | awk '{print $1}' | grep -o '[0-9]*$' | jq -s max) # Gets total number of existing load-balancers to avoid data-centre name duplications
max_consul=$(gcloud compute instances list | grep $consul | awk '{print $1}' | grep -o '[0-9]*$' | jq -s max) # Gets total number of existing consul-masters to avoid consul server name duplications
k=1
l=1
echo "THIS MAY TAKE A WHILE, PLEASE BE PATIENT WHILE THE COMMANDS ARE RUNNING..."

./sidekicks/clean-instance-group.sh # Initiates instance-group cleanup process

for i in $(echo $zone_name | sed "s/,/ /g"); do
    printf "\nPopulating $i \n"
    cmd="gcloud -q compute instances create $consul$((max_consul + k)) --image $image --machine-type $consul_machine --zone $i --tags http-server --service-account $serv_account --scopes cloud-platform --metadata-from-file startup-script=./startups/consul-master.sh" # Creates a consul-master instance for each data-centre
    progress;

    cmd="gcloud -q compute instances create $lb$((max_lb + k)) --image $image --machine-type $lb_machine --zone $i --tags http-server --service-account $serv_account --scopes cloud-platform --metadata-from-file startup-
```

```
script=./startups/lb.sh" # Creates a load-balancer instance for each data-centre
    progress;

    cmd="gcloud -q compute instances create nat-gateway-dc$((max_lb + k)) --network default --can-ip-forward --image $image --machine-type $consul_machine --zone $i --tags nat --service-account $serv_account --scopes cloud-platform --metadata-from-file startup-script=./startups/nat-gateway.sh" # Creates a nat-gateway instance for each zone
    progress;

    cmd="gcloud -q compute routes delete no-ip-internet-route-dc$((max_lb + k))" # Deletes the forwarding-route for the the specific zone if exists
    progress;

    cmd="gcloud -q compute routes create no-ip-internet-route-dc$((max_lb + k)) --network default --destination-range 0.0.0.0/0 --next-hop-instance nat-gateway-dc$((max_lb + k)) --next-hop-instance-zone $i --tags no-ip-dc$((max_lb + k)) --priority 800" # Creates a forwarding-route for all the worker instances that connects trhough nat-gateway instance
    progress;

    max_inst=$(gcloud compute instances list | grep $inst-dc$((max_lb + k)) | awk '{print $1}' | grep -o '[0-9]*$' | jq -s max) # Finds the number of existing worker instances in the data-centre
    l=1
    for j in $(eval echo {1..$inst_number}); do
        cmd="gcloud -q compute instances create $inst-dc$((max_lb +k))-$((max_inst + l)) --network default --no-address --image $image --machine-type $inst_machine --zone $i --tags no-ip-dc$((max_lb + k)) --service-account $serv_account --scopes cloud-platform --metadata-from-file startup-script=./startups/worker-$variant/worker-$variant.sh" # Creates worker instances with nat-gateway tags and service account properties
        progress;
        let "l++"
    done
    let "k++"
done

./sidekicks/create-instance-group.sh # Initiates instance-group setup process
./sidekicks/dns.sh # Initiates cloud DNS setup process
```

Appendix F

Performance Evaluation Scripts

F.1 data-collector.sh

```
#!/bin/bash

# Copyright: Freeware (any user can download or distribute
    this script)
# Run the code at your own risk. Follow the instructions in
    README.
# Author: Samiul Saki
# Email: samiulsaki@gmail.com

# Script: Testing

declare -a lb=('lb-dc1' 'lb-dc2' 'lb-dc3')

for argh in ${lb[@]}; do
        lb_ip=$(gcloud compute instances list | grep $argh |
            awk '{print $5}') # Finds all the load-balancer's
            IP addresses
        rtime=$(eval "curl -sSL 'http://$lb_ip/haproxy?stats;
            csv;norefresh' | cut -d "," -f 2,61 | column -s, -t
             | grep worker | awk '{print \$2}' ") # Curl the
            HAProxy stats in csv format running in the load-
            balancer
        sum=0
        for i in $rtime; do # Sum them together in a loop
                sum=$((sum + i))
        done
    eval "var=$(echo $argh | sed "s/lb-//g" | sed "s/.csv//g")
        " # Creates dc variables
    avg_resp=$(( sum / $(echo "${rtime}" | wc -l) )) #
        Calculates average response time of the laod-balancer
    div="1000"
    eval "${var}_resp=$(echo "($avg_resp) / $div" | bc -l)" #
        Convert the response time in seconds and insert in the
        dc_resp variable
    eval "${var}_worker=$(echo $(echo "${rtime}" | wc -l))" #
        Creates and add the number of workers in dc_worker
        variable
done

```

```
printf "`date '+%Y%m%d-%H%M%S'`,$dc1_resp,$dc1_worker,$dc2_resp,$dc2_worker,$dc3_resp,$dc3_worker\n" >> ./file.csv # Appeneds the data to the csv file
```

F.2 ewma.py

```
#!/usr/bin/python

import os, sys, re
import numpy as np
import time
from datetime import datetime
import pandas as pd
import matplotlib.pyplot as plt

file=sys.argv[1]

fmt = "%Y%m%d-%H%M%S"
df = pd.read_table(file, sep=',', skiprows=1, skipfooter=0, engine='python',
                   names=['Ptime', 'DC1_Resp', 'DC1_Worker', 'DC2_Resp', 'DC2_Worker', 'DC3_Resp', 'DC3_Worker'],
                   parse_dates=True,
                   date_parser=lambda s: datetime.strptime(s, fmt),
                   index_col=0)


def plot(DC1=False, DC1_raw=False, DC1_ma=False, DC1_ewma=False, DC2=False, DC2_raw=False, DC2_ma=False, DC2_ewma=False, DC3=False, DC3_raw=False, DC3_ma=False, DC3_ewma=False, window_span=None, periods=None, show=False, savefig=False, scheme='', label=''):
    plt.figure()
    plt.title(label)
    if DC1:
        dc1_resp = df['DC1_Resp']
        if DC1_raw:
            dc1_resp.plot(label='Raw data DC1')
        if DC1_ma:
            roll1 = dc1_resp.rolling( center=False, window=window_span, min_periods=periods).mean()
            roll1.plot(label='Moving Average DC1', linestyle='--', linewidth=1.2)
        if DC1_ewma:
            ewma1 = pd.ewma(dc1_resp, span=window_span, min_periods=periods)
            ewma1.plot(label='EW Moving Average DC1', linestyle='--', linewidth=1.2)
    if DC2:
        dc2_resp = df['DC2_Resp']
        if DC2_raw:
            dc2_resp.plot(label='Raw data DC2')
        if DC2_ma:
```

```python
            roll2 = dc2_resp.rolling(center=False, window=window_span, min_periods=periods).mean()
            roll2.plot(label='Moving Average DC2', linestyle='--', linewidth=1.2)
        if DC2_ewma:
            ewma2 = pd.ewma(dc2_resp, span=window_span, min_periods=periods)
            ewma2.plot(label='EW Moving Average DC2', linestyle='--', linewidth=1.2)
    if DC3:
        dc3_resp = df['DC3_Resp']
        if DC3_raw:
            dc3_resp.plot(label='Raw data DC3')
        if DC3_ma:
            roll3 = dc3_resp.rolling(center=False, window=window_span, min_periods=periods).mean()
            roll3.plot(label='Moving Average DC3', linestyle='--', linewidth=1.2)
        if DC3_ewma:
            ewma3 = pd.ewma(dc3_resp, span=window_span, min_periods=periods)
            ewma3.plot(label='EW Moving Average DC3', linestyle='--', linewidth=1.2)
    os.system('clear')
    plt.legend(loc='best')
    plt.xlabel('Time (in secs)')
    plt.ylabel('Average Response Times (in secs)')
    plt.tight_layout()
    if show: plt.show()
    if savefig: plt.savefig('./plots/plots_testbed_' + str(scheme) +'.png')
    plt.close()

plot(
    DC1=True, DC1_raw=False, DC1_ma=False, DC1_ewma=True,
    DC2=True, DC2_raw=False, DC2_ma=False, DC2_ewma=True,
    DC3=True, DC3_raw=False, DC3_ma=False, DC3_ewma=True,
    window_span=100, periods=1, show=False, savefig=True,
    scheme='[variant]', label='Test-Bed: Response Times - [variant]'
)
```