An Adaptive System for Allocating Virtual Machines in Clouds using Autoregression

by

Jasmeet Singh

A thesis submitted to the
Faculty of Graduate and Postdoctoral Affairs
in
partial fulfillment of the requirements for the degree of
Masters of Applied Science in Electrical and Computer Engineering

Department of Systems and Computer Engineering
Carleton University
Ottawa, Ontario
Canada

Abstract

This thesis proposes an adaptive system to allocate virtual machines in a cloud environment to reduce clients' waiting time while reducing the idle resources for the service provider. Further, the thesis demonstrates the viability of the proposed system via a prototype built using the Citrix XenServer and a machine learning algorithm which makes the system capable of working with minimum human interactions. The proposed architecture is designed in collaboration with and based on the requirements of DLS Technology so that they can migrate their flagship product (vKey) to a cloud environment keeping security and performance as a priority. The incoming requests from clients are handled by a pool manager which takes smart decisions thus making the user experience seamless. A performance analysis of the prototype is carried out to prove the effectiveness of the proposed strategies.

Acknowledgments

I, Jasmeet Singh, take this opportunity to express my deep sense of gratitude to all the people who have developed me in the successful completion of this thesis. As no task is a single man's feat, various factors, situations, and person integrate to provide the background to accomplish a task. Several persons with whom I have interacted significantly helped me to the successful completion of this thesis.

I own a deep sense of gratitude to Professor Marc St-Hilaire and Professor Shikharesh Majumdar. I can't convey my thanks in words for there tremendous support, help, and motivation throughout the process.

I would like to owe the same gratitude to Eric She, Jordan Kurosky and Sibyl Weng from DLS Technology, Ottawa for believing in me with their confidential work and providing enormous help. I would also like to convey my thanks to Hindal Mirza from Ontario Centers of Excellence (OCE) for providing the funding for the project.

I would like to thank my parents for helping me and providing me with support and encouragement throughout my academic achievements even within miles of distance between us.

My special thanks go to my friend Amarjit Singh Dhillon who during the last two years was always around me for a study or personal talks and has shared my ups and downs. My thanks extend to my friends Manpreet Dhillon, Srikant Kumar and Jay Bhuchhada for their continuous support.

Table of Contents

List of Figures

List of Tables

List of Algorithms

List of Abbreviations

API	Application Program Interface
ARIMA	Autoregressive Integrated Moving Average
AWS	Amazon Web Services
BIOS	Basic Input-Output System
BS	Baseline System
CPU	Central Processing Unit
DB	Database
DMZ	De-Militarized Zone
EC	Elastic Compute
FIFO	First In First Out
GB	GigaByte
I/O	Input/Output
IaaS	Infrastructure as a Service
IRAP	Industrial Research Assistance Program
IT	Information Technology
JSON	JavaScript Object Notation
KNN	K Nearest Neighbors
KVM	Kernel-Based Virtual Machine
MB	MegaByte
NSERC	Natural Sciences and Engineering Research Council of Canada
OCE	Ontario Centers of Excellence

OCR	Optical Character Recognition
OS	Operating System
PaaS	Platform as a Service
PAS	Proactive System
QEMU	Quick Emulator
RAM	Random Access Memory
RPC	Remote Procedure Call
RS	Reactive System
SaaS	Software as a Service
SLA	Service Level Agreement
SR	Storage Repository
STM	Service Time Manager
USB	Universal Serial Bus
VDI	Virtual Disk Image
vDM	vKey Device Manager
VIF	Virtual Network Interface
VMM	Virtual Machine Monitor
VM	Virtual Machine
XAPI	Xen Application Program Interface
XML	Extensible Markup Language

Chapter 1

Introduction

1.1 Motivation for the Thesis

This thesis was carried out as part of a research collaboration among DLS Technology, Carleton University and the Ontario Centers of Excellence (OCE). DLS is an IT company located in Ottawa, Canada which has developed a software called vKey. vKey is a secure platform used by various organizations such as the government of Canada to "access their organization's network, applications and data from any host computer without changing how they work, and without compromising corporate network security" [1]. vKey is first loaded onto any bootable media device such as USB stick, hard disk and microSD, then it is plugged into a personal computer and the system is rebooted. vKey acts like a deployed virtual laptop which provides a trusted secure remote access by bypassing the host operating system and key loggers.

Currently, vKey is only accessible from a physical resource that needs to be physically plugged into a computer system and requires a reboot each time it is used. With the goal of improving the usability of the product, the company decided to migrate vKey to a cloud environment. By doing so, the company is increasing the ease of using the product, the user experience, and the ability to provide a high-quality service while concealing the user data. Along with meeting the Service Level Agreement (SLA) standards, the security of the data on the Virtual Machine (VM) is also given high priority.

While moving vKey to a cloud environment, the company was facing critical issues such as providing secure remote access to users via vKey, optimizing the use of cloud servers, decreasing the wait time for clients and reducing the amount of idle resources. Several commercial solutions and virtualization technologies such as Bochs, Microsoft Hyper-V, etc. are available to achieve some of these goals, but each one comes with its own downsides such as specialized hardware, cost, and performance issues.

Resource allocation has been a subject of discussion for many years in several computing areas such as data center management, operating systems, and grid computing. Many researchers have described resource allocation as a mechanism which ensures that the needs of the application are properly taken care by the provider who is providing the infrastructure. A resource allocation system should take into consideration the present status of each resource, the number and the size of incoming requests and should use smart algorithms to better allocate physical and virtual resources to the applications. A good resource allocation system will potentially be able to accommodate more users and will also minimize the operational cost.

Allocating a virtual machine to a client is a two-step process. In the first step, the request for virtual machine provisioning is accepted and is placed on the server. The second step is to allocate resources to a virtual machine depending on the client demands. The complexity of the allocation depends on the number of virtual machines that needs to be allocated and the number of available servers. The virtual machine allocation algorithm ensures an efficient and cost-effective allocation of resources.

Most of the work that has been done in this thesis focuses on reducing the waiting time for customers and the amount of idle resources. Moreover, we also have to consider whether all the requested virtual machines are getting allocated and whether the service provider is able to serve more requests in a particular time-frame. To achieve the above mentioned goals and based on the above evolution of vKey and current technologies, DLS Technology was looking for a research collaboration to come up with and develop a prototype to efficiently and cost-effectively allocate virtual machines to users using the cloud environment. As a solution, we propose a new technique based on prediction, which learns and configure the system to accommodate the incoming clients within a specific time-frame.

1.2 Research Objectives

Based on the problem statement described above, the main goal of this thesis is to develop an adaptive system to allocate virtual machines in the cloud to provide secure access to end users using vKey. The key objectives of this research are presented below:

- Propose a new model to allocate virtual machines to users. The model should be able to:
  - reduce the client's waiting time. The system should maintain a pool of pre-created virtual machines such that the waiting time in the queue is minimized before using the service.
  - use a prediction algorithm to understand the clients' behavior and proactively reconfigure the system to accommodate the incoming client requests

    to reduce the waiting time for the clients and reduce the amount of idle resources in the pool.

    - destroy the memory and the virtual disk at the end of each user session to avoid compromising the corporate network security.
    - provide the clients a fresh copy of the virtual machine (with vKey installed on it) each time upon request.

- Develop, implement and test a prototype achieving the above mentioned subobjectives so that it can be successfully delivered to DLS Technology.

1.3 Proposed Solution

The goal of the thesis is to design an adaptive system which manages the allocation of virtual machines and learns the user behavior over time. This satisfies the research and the enterprise needs of the high-quality user experience with, lower cost and higher resource efficiency. The simplest way of allocating a virtual machine is to create and start a VM upon the client request. This solution is probably not the best practice as it requires a finite amount of time to create and start a new VM and results in a long waiting time for the users. Therefore, to overcome the issues of the previous approach, we designed a new model, referred to as "pool manager" to meet the client demands. The pool manager uses a machine learning algorithm to predict the number of virtual machines required to handle the client requests in a specific time period. Then, it precreates the VMs with vKey and allocates them to users upon request. Once, the user ends its session, the VM is destroyed (including the disk image and the memory allocated to it). This security measure ensures that when a new request comes in, a brand new VM is assigned to it.

1.4 Contributions of the Thesis

- The main contribution of the thesis is a new model (referred to as pool manager) which can proactively create and allocate VM to users.
- A prediction algorithm is used to learn the behavior of the users coming into the system, based on which the number of virtual machines required in the next specific time period is calculated. Based on this forecasted values, the system is reconfigured to accommodate the incoming clients.
- A destroyer algorithm is designed such that once the user ends a session, the virtual machine is deleted from the pool along with its disk image and memory. This concept is implemented as it was a request from the company so that users information is lost completely once a session is terminated.
- A proof of concept prototype based on the Citrix XenServer to showcase the working and the effectiveness of the proposed system. Also, performance analysis leads to a number of insights among the three systems and workload parameters.

1.5 Thesis Outline

The rest of the thesis is divided into five chapters as described below.

Chapter 2: discusses the background, the related work and theoretically explains concepts related to virtualization. This thesis mainly focuses on Citrix products such as

XenServer, Xen center, and the Xen API.

Chapter 3: discusses the methodologies proposed with the aim to satisfy the company and research requirements. The first prototype is a simple client-server architecture: the client comes in with a request, then the system creates a new VM and allocates it to the user. In the second prototype, a new pool logic algorithm has been introduced. It pre-creates a number of VMs and stores them in a pool, then allocates a VM when a user request arrives. The third prototype is an adaptive system which learns and predicts the behavior of incoming arrivals over time and adjusts the number of VMs in the pool accordingly. This, in turn, decreases the amount of idle resources.

Chapter 4: discusses the performance analysis of the three proposed systems including different performance metrics, experimentation results, and the comparison of the implemented systems.

Chapter 5: concludes the thesis and discusses the possible directions for future work.

Chapter 2

Background and Related Work

This chapter begins by outlining the concept and types of virtualization. Then, it provides an overview of cloud computing. Further, it explains the different components of Citrix XenServer and how they are related to a cloud environment. The chapter then focuses on machine learning and the autoregression model. Finally, the chapter is concluded by outlining the existing work on various techniques to allocate virtual machines in the cloud.

2.1 What is Virtualization

As DLS Technology is in the process of moving vKey to the cloud environment, it is important to clearly understand the concepts of virtualization.

Virtualization, in the context of this thesis, concerns running multiple operating systems on a single machine but sharing all the hardware resources. In other words, virtualization could be defined as the separation of logical operations from the physical environment [2]. To understand better, let us take an example of virtual memory. Virtual memory is an extension to the system memory that is derived from the hard drive instead of Random Access Memory (RAM). It means that if the system runs out of memory during an operation, the virtual extension can come to the rescue and can be used to keep the system running [3]. This might be put into effect while designing the system to overcome a scenario where several clients join the system at the same time which might make the hardware run out of memory.

There are numerous types of virtualization techniques, but to keep within the scope of this thesis, we will be studying only about server virtualization. Figure 2.1 portrays the overview of the architecture of a virtualized environment. The architecture is divided into three levels. The lowest level is the hardware level that contains the physical server (Central Processing Unit (CPU), memory, storage, network files) and the host Operating System (OS). The second level, which sits on top of the host OS, is the virtualization layer. The virtualization level consists of the virtual machines and the hypervisor which will be described momentarily. The top level in the architecture is the user level where all the applications and jobs that user wants to run on the guest OS, without the knowledge of the host OS.

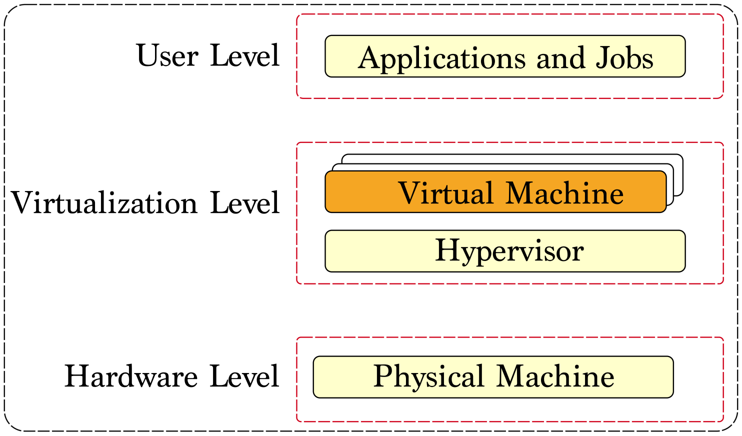


Figure 2.1: Overview of the virtualization architecture

A virtual machine is a computer image file that behaves like a personal computer system with all the requirements that a user demands. Multiple OSs can run simultaneously on the same physical server side-by-side using a software called hypervisor. The hypervisor decouples the virtual machine from the host and dynamically allocates the resources to individual virtual machines based on the requirements. Furthermore, each virtual machine provides its own virtual hardware (resources) that includes CPU, hard drive, memory and network interface. In the next step, the virtual hardware is mapped onto the physical hardware which in return reduces the number of physical servers required to accommodate a large number of users and therefore reducing the operational cost for the servers. Along with cost savings related to buying new host machines, virtualization provides great security at the hardware level with the concept called isolation. It is also possible to save the state of an entire virtual machine to a file and migrate it across the server pool.

2.1.1 Types of Virtualization

There are many virtualization technologies available today and that could be used appropriately depending on the needs of the organization. Server virtualization is broadly divided into three groups: OS level virtualization, paravirtualization and full virtualization. They are described below.

2.1.2 OS Level Virtualization

This is the common type of virtualization technique that is accepted broadly in the market. In this model of virtualization, every single virtual machine runs as an isolated instance of the operating system within the virtualization software. The software that is used for virtualizing runs on the main hardware and is generally referred to as a host

OS and creates a virtualized guest OS to provide the execution environment for the applications. Thus, it means the hardware is not virtualized specifically, but only provides the needed services to the guest OS. The products that use OS level virtualization are Oracle virtual box, parallel workstations, and VMware. Figure 2.2 gives an overview architecture for the OS level virtualization.

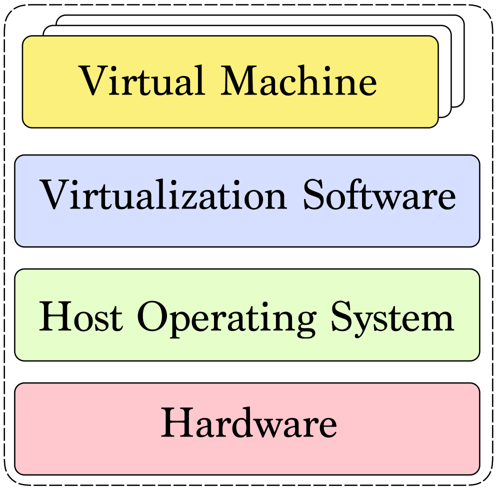


Figure 2.2: OS level virtualization architecture

2.1.3 Paravirtualization

Before learning about paravirtualization, we need to understand about hypervisor-based virtualization, as both para and full virtualization based architectures, have foundations on this. A hypervisor is a software that manages a virtual platform and operates between the physical layer and the host operating system. This software is also known as a Virtual Machine Monitor (VMM). It provides the features and services that are required for unobstructed operation and execution of virtual machines. The hypervisor identifies traps and responds to protected CPU instructions that are made by each virtual machine [4]. Figure 2.3 showcases the hypervisor-based virtualization. Along with handling queuing and dispatching of the hardware requests, there is an administrative operating system (host OS) with the motive to administer the virtual machines.

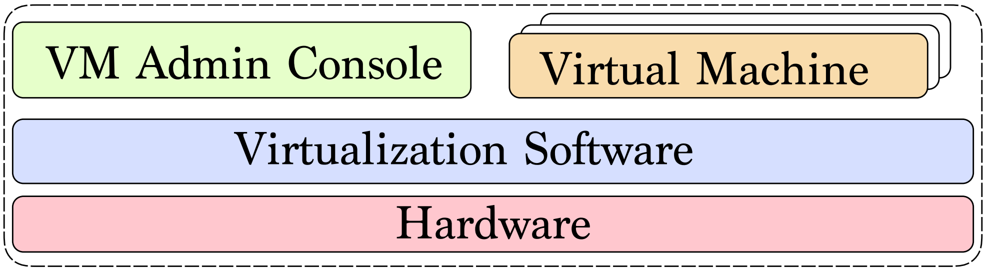


Figure 2.3: Hypervisor virtualization

Hypervisors can be further divided into three types [5]:

- Type 1 - Hypervisors of type 1 deals with the virtualization of the hardware and the hypervisor VMM is placed on top of the physical hardware and runs directly on it (there is no host OS).
- Type 2 - in this type, the hypervisor is placed on the top of the host OS and is also called process virtualization.
- Hybrid - it handles the hardware virtualization by emulating the entire hardware, where the virtual systems then lay above the host OS and hypervisor.

Now, that we have the basic idea of what exactly hypervisor virtualization is, let us understand paravirtualization. The paravirtualization approach was designed by a company named Xen and later was adapted by many other companies and made available as open source. Para comes from the Greek word which means “alongside” or “beside” in paravirtualization. As shown in Figure 2.5, Hagen Von [6] in his book explains the

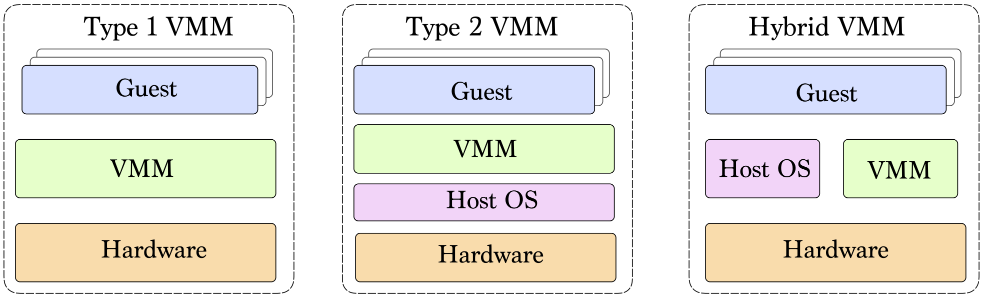


Figure 2.4: Types of hypervisor

hypervisor as a very small and compact in size, thus allowing it to run directly on the hardware without any significant overhead. Paravirtualization does not have a clear division between guest and host OS, rather it uses a trusted guest host referred to as Domain0.

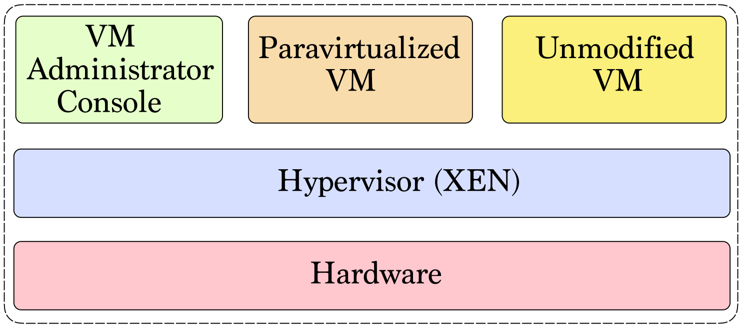


Figure 2.5: Architecture of paravirtualization

Domain0 is a customized Linux kernel that manages the virtual machine manager, builds the additional virtual machines and controls them. It eliminates most of the trapping and emulation overhead which are related to software implementation. This is the reason why XenServer uses paravirtualization and therefore another reason we decided to use XenServer over any other technology.

2.1.4 Full Virtualization

Full virtualization has a very similar approach to paravirtualization. Full virtualization uses a type 1 hypervisor, which is an unmodified host OS that lies on the top of the hypervisor. In this, the guest OS remains the same as the host OS with the capability to accept any changes in the physical hardware.

In the full virtualization model, the hypervisor traps the machine operations that the OS use to read or modify the status of the system or to perform input-output operations. Once the hypervisor has trapped the information, it emulates the operations in the software and returns a status code which is the same as what a real hardware would deliver. As it traps the system information, it uses a significant amount of memory and processor which degrades the operational speed. Figure 2.6 gives an overview of the full virtualization model.

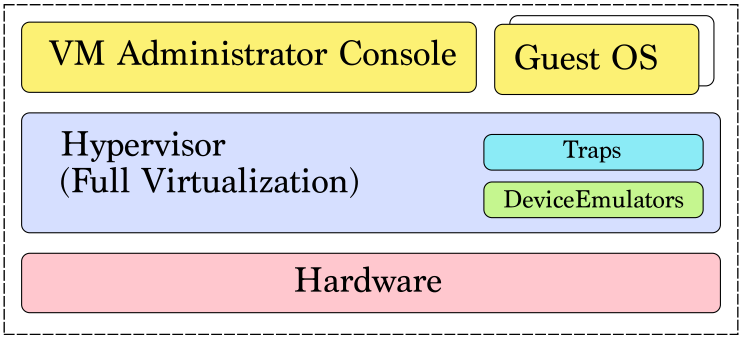


Figure 2.6: Architecture of full virtualization

2.2 Overview of Cloud Computing

From all the existing definitions used for the term cloud computing, the one which consolidates all the aspects of cloud computing is given by Gartner which defines cloud computing as “an elastic computational model where its IT related compatibilities are delivered as a service to multiple external customers” [7].

Cloud computing is useful when services like storage, computation and data management do not require the knowledge of the end user or the physical location and the configuration of the system that might be using it. In other words, it is free from all the dependencies of traditional computers. The consumerization of IT and the emergence of cloud services like Amazon Web Services (AWS), Elastic Compute (EC), Rackspace, and Google have increased the capacity of services being delivered and consumed. Thus, cloud computing has satisfied these needs by adding capabilities on demand without investing in new infrastructure and licensing and this results in the reduced cost of operation.

How does cloud computing work? Cloud computing providers deliver the application to the user utilizing a very old technology called the Internet, which is accessible from a web browser, with all the information and data stored on remote servers in data centers. Cloud computing consists of services which are made available through data centers which are the point of contact for the users for their computing requirements.

In a diverse prospective, we need to respond faster and be able to accommodate changes in the software industry based on the customer demands and growth. With this growing demand for cloud computing, virtualization plays an important role. It is also said that cloud computing is a technological evolution of virtualization platforms.

Cloud computing inherits some characteristics from the traditional computing models. Firstly, autonomic computing designates distributed computational systems capable of adapting to the unpredictable changes and, at the same time, hiding the complexity from the users. Second, the client-server model states that application differentiates between service providers from service requestors. Third, the method in which massive virtual computers, a cluster of networks, or loosely coupled computers act as a single entity to perform very large tasks. Now, that we have an understanding of what cloud computing is, let us go deeper and study its architecture. Most of the clouds today are built on top of modern centers that provide Infrastructure as a Service (IaaS), Platform as a Service (PaaS), and Software as a Service (SaaS) and these services are provided as utilities. All these services are different layers of the cloud pyramid and are built upon one another, creating a larger whole. It is a possibility that all the layers might be dependent on each other, but not necessarily require the interdependency, as a matter of fact, each layer stands by itself.

The foundation of any cloud computing environment begins with data centers, which provides the hardware to the cloud on which it will run. Built on the top of the data center layer is the IaaS layer. This layer is responsible for virtualizing the computing power, storage, and connectivity to the data centers. The cloud infrastructure providers give users a complete control over the virtualized resources and make them believe as if they are the only ones using the resources. Next, on top of the IaaS layer sits the PaaS layer. The PaaS layer is often known as the cloudware which provides the development platform for application designing, development, testing, deployment to monitoring and hosting on the cloud. The top layer of the cloud pyramid is SaaS. This is the layer where the end user works on the services that are provided to him upon request, which is usually done in a browser. SaaS saves the user from the cost of additional licenses or troubles of software deployment and maintenance. The software is shared by many users and is automatically updated from the cloud.

So far, we have discussed basic concepts and the architecture of cloud computing. How clouds can be implemented in different ways to better suit user requirements is discussed next. There are three ways in which clouds can be deployed and are defined below in some detail and Figure 2.7 gives a pictorial view of the cloud. First, is the type which we all are aware of and is widely used in today's IT world, is termed as a public cloud. Public cloud describes the cloud computing in a traditional way, where resources are allocated dynamically to the general public over the Internet with the help of web applications or web services. Providers of these services are located off-site and use pay per go for computations. The second category is private cloud, it is an infrastructure operated and managed by a single organization exclusively and can be managed internally or via third party vendors. A private cloud provides better security at the expense of the cost of buying, building, and managing the cloud. Finally, the third category of the cloud is the least known and is called a hybrid cloud. It is composed of two or more clouds that could be public or private, that are kept distinctive but are united together which offers the combined benefits of public and private models.

2.3 Components of Citrix XenServer

This section explains the various technical pieces of the Citrix XenServer that will be used in this thesis.

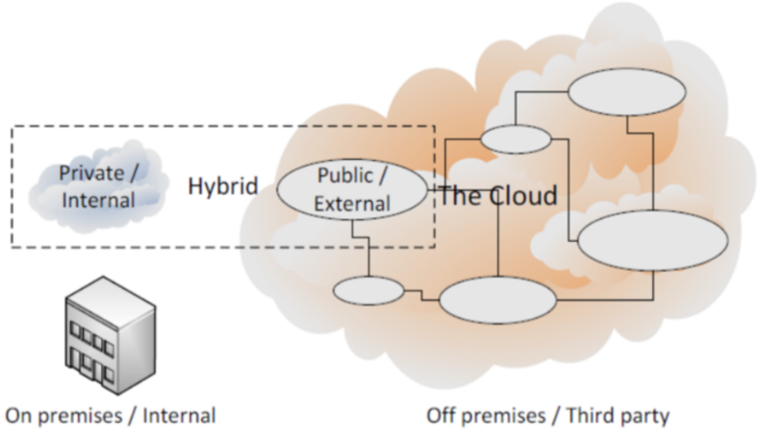


Figure 2.7: Types of cloud computing [8]

2.3.1 Citrix Xen

The Xen came into the picture in 2003 [9] and was a research project at the University of Cambridge. The Xen hypervisor has a very different architecture from conventional hypervisors like Kernel-Based Virtual Machine (KVM)[10] and Quick Emulator (QEMU) [11]. KVM is a kernel module that uses the host's OS (Linux) as a hypervisor, whereas Xen is itself a hypervisor.

The Xen hypervisor is an open source standard system widely used for virtualization in the industry. It is powerful, efficient, secure and these features make it a perfect fit for virtualization of currently used CPU architectures. The Xen hypervisor currently supports the following operating systems: Linux, Solaris, Windows.

2.3.2 Xen Virtualization

Xen uses the technology called paravirtualization as already discussed in this thesis in Section 2.1.3. From the previous section, Xen uses a hypervisor which sits on the top of the hardware and is executed directly from there. The primary task of the hypervisor is to partition the memory and schedule them for the guest users. The execution of all the computations of the guests are handled by the hypervisor, but it does not handle the input-output operations or peripheral devices and has no knowledge of the physical drives.

Domain0, or Dom0 in short, is a special virtual machine which is responsible for handling all the Input/Output (I/O), disk storage and all the operating system resources. Dom0 is basically a modified Linux kernel that is required for Xen to execute any guests operations. Traditionally, in a KVM architecture, only the hardware virtualization was added to provide support to the kernel. In contrast, Xen runs directly on the hardware and acts as a mediator that handles things like input-output, storage, etc.

DomainU, or DomU, is the guest that runs on the Xen hypervisor and is a regular virtual machine. In Figure 2.8, an important thing to notice is that DomU is placed in parallel with Dom0 to make usage of processor fair among the virtual machines, as host is treated equally by the guests. The current credit scheduler[1] assigns equal weights to both Dom0 and DomU, thus giving them equal priority. When the Xen was invented, it was using paravirtualization to support a guest operating system as mentioned in Section 2.1.1. However, with the advancements of IT, Xen can now be fully virtualized but this might cause some degradation in performance [6]. The difference in paravirtualization is that the guests used special drivers which gave them a sense of being virtualized. With full virtualization, guests can operate unmodified and are not aware of being virtualized. Unmodified guests mean using the operating systems that are not modified

[1] https://www.wiki.xen.org/wiki/credit-scheduler

for usage with the virtual machine. To achieve full virtualization, Xen uses a device model which provides an emulated machine to the guests which is a simple version of QEMU. This emulated machine is responsible for handling the I/O operations, storage.

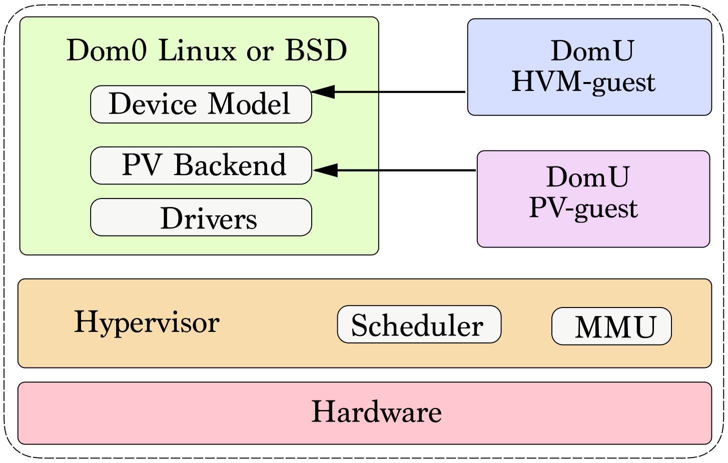


Figure 2.8: Architecture showing Xen virtualization

2.3.3 Citrix XenServer - A Cloud-optimized Server

Citrix XenServer is an open source platform for cloud, server, and desktop virtualization infrastructure [12]. It enables users to deploy Windows and Linux machines rapidly, and manage them efficiently. Citrix XenServer can also manage the storage and network devices of the virtual machines from a single console.

Citrix XenServer uses VMM in a full virtualization technique, to provide an image of the entire system that includes virtual Basic Input-Output System (BIOS), virtual memory, and virtual devices to a virtual machine that a guest will be using. The VMM is also responsible for creating and maintaining the data structure for the virtual components. Every time a virtual machine accesses these components, the VMM updates the data structure [8].

Figure 2.9 defines the XenServer architecture and how the access is maintained between the physical host and Dom0. The hypervisor is the first element of the server that is loaded from the local storage, which then establishes an interface with the compute in order to provide services to virtual machines. The first virtual machine on the system is called dom0, and then the guest virtual machines are created called domU, both have been discussed in Section 2.3.2. Dom0 provides the guest VMs access to host hardware via Linux device drivers. The drivers then provide an interface with XenServer process, resulting in a virtual device interface using a split-driver model. Further, QEMU is responsible for emulating the hardware components, providing network and the disk access. Finally, Xen Application Program Interface (XAPI), a toolstack, binds everything together.

To summarize, Citrix XenServer is an enterprise-ready server virtualization system and together with a cloud computing platform, it provides a range of guest operating systems with the network and management tools within a single, tested and open source installable image. With these features, it consolidates server workloads, power saving, cooling, management and the ability to adapt to the changing IT environment [6].

2.3.4 Xen API

The Application Program Interface (API) for the XenServer uses a set of remote procedure calls and has a format based on XML-RPC. These API calls remotely configure and control the virtualized guests running in a XenServer pool. The references to the API use classes and objects. The class is a hierarchical namespace and an object is an

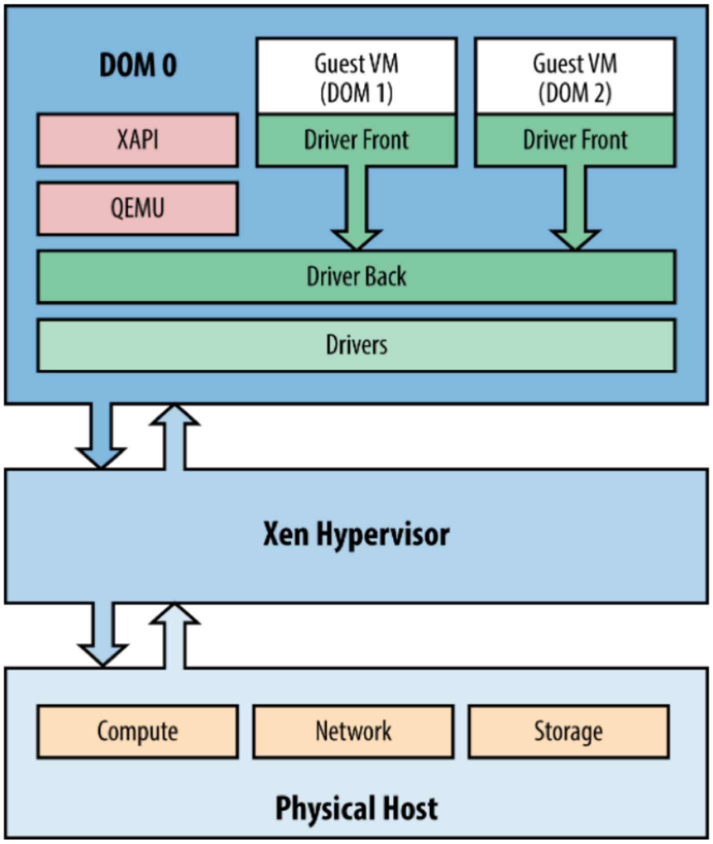


Figure 2.9: XenServer [13]

instance of a class with the fields set to specific values. The relation between the various classes that we have used in the implementation of our system is pictorially represented in Figure 2.10. In that figure (obtained from [14]), a rectangle represents an object class and an ellipse represents an instance of a specific class.

A brief discussion of a few basic classes of a Xen API that are used in the system are presented next.

- Session: It is used to establish a session between the user and server.

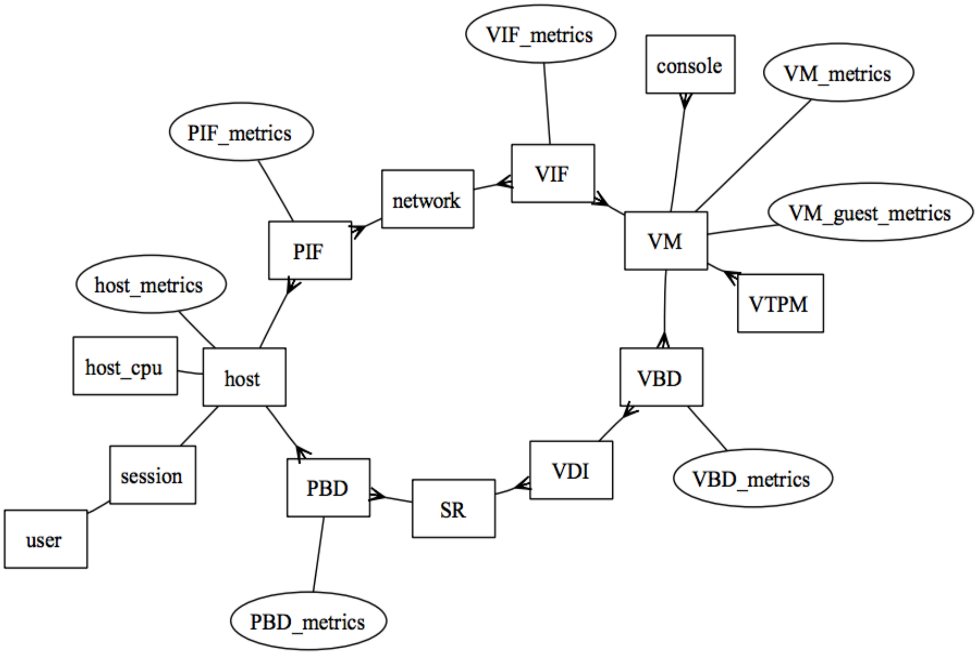


Figure 2.10: Xen API [14]

- VM: It identifies a guest virtual machine.
- Host: Represents a physical host i.e. the Citrix XenServer.
- VIF: It is used to create the virtual network interface of the guest virtual machine.
- SR: It represents a storage repository.

Along with these classes, there are various operations or functions associated and we have used them for successful completion of the various tasks. The basic functions used in the thesis are:

- Copy: to create a virtual machine.

- Destroy: to delete a virtual machine including its storage repository, disk image and memory associated with it.
- Reboot: to reboot the virtual machine.
- Start: to start a virtual machine upon allocation to the user.
- Shutdown: to halt a virtual machine before being destroyed.

2.4 Machine Learning

Machine learning is a kind of artificial intelligence. It is a learning process of computers without any programming interaction with the computers where they learn by external stimuli, also called as 'Unsupervised Learning'. The term 'Machine Learning' was coined by an American pioneer Arthur Samuel, who studied computer gaming and artificial intelligence. The study of machine learning involves understanding and building algorithms which are capable of learning and extracting information from the data provided. These can also predict using the data and can overcome the traditional program instruction methodology. Machine learning can be used in various real-life scenarios where providing written programmable instructions is not a feasible option or not possible otherwise. Some examples enlisted are intrusions in a network, Optical Character Recognition (OCR), computer vision, etc.

With a progressive use of technology among the people, massive amounts of data are available for access which is termed as 'Big Data'. The government and companies have access to big data, but due to a lack of resources, they are incapable of making a beneficial use out of it. Machine learning, a method of artificial intelligence, is becoming popular for utilization to extract and use data from big data.

The process of machine learning involves generalizing from its experience and by reading data, to enhance the machine's capability to react accurately to any new experience through the use of training on the data over time. There are three main types of machine learning [15] which are described below and shown in the Figure 2.11.

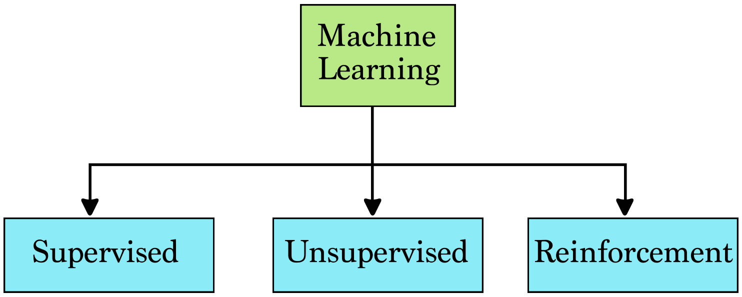


Figure 2.11: Types of machine learning

2.4.1 Supervised Learning

There is a target or an outcome variable which is predicted using some pre-existing independent variables. With the use of these independent variables, a mapping is done from the inputs to the required outputs. This methodology continues until the machine reaches a level where it does this mapping on its own quite accurately from the training data. Some instances of supervised learning are decision tree, K Nearest Neighbors (KNN), regression, etc.

2.4.2 Unsupervised Learning

There is no specific target or outcome variable in this kind of learning. This is used to cluster the population into distinctive groups (also called data set) and then use a suitable model to train on the data in order to forecast an outcome. For example K-means algorithm, etc.

2.4.3 Reinforcement Learning

This machine learning methodology is used to make the machine capable of making some decisions on its own. The machine in this method is exposed to external stimuli or environment where it trains itself independently in different situations through a trial and error method. The machine keeps on saving and learning from the previous experience and tries to react in the best way possible using this experience. An example of this type of learning is a Markov decision process.

2.5 Autoregression

Autoregressive models are a fundamental class of time series models. Time series is a sequential set of data points measured typically over successive times [16]. Time series are of two types: univariate where only one variable is considered in the records and multivariate when more than one variables are considered in the records. Autoregression is a widely used linear time series model [17] and is defined as a model that predicts the future value of the next time step based on the values from previous time steps as the input to the regression Equation (2.1), where c is a constant, e_t is the white noise and y_t are the lagged values (also called past series values). The autoregression process is an example of a stochastic process, that has degrees of uncertainty and randomness built-in. The randomness means we might be able to predict the future values pretty easily with the past data but will never achieve an accuracy of 100 percent.

$$y_t = c + \phi_1 y_{t-1} + \phi_2 y_{t-2} + \cdots + \phi_p y_{t-p} + e_t \tag{2.1}$$

Autoregressive models take the advantage of the fact that time series data is sequential in nature and it is very important to preserve the sequence of the data to make better predictions. The model uses the weighted sum of the past values to forecast the future values. In Equation (2.1), the coefficients ϕ_1, ϕ_2, ..., ϕ_p define the importance of the lagged values. Lets take an example by rewriting the equation with some arbitrary numbers to make things more clear:

$$y_t = c + 0.2y_{t-1} + 0.8y_{t-2} + \cdots + 0.4y_{t-p} + e_t \tag{2.2}$$

Equation (2.2) conveys that our prediction y_t is heavily dependent on the second most recent value y_{t-2} and least dependent on the most recent value y_{t-1}.

The correlation between the two coefficients of the lagged values is defined as autocorrelation. Autocorrelation is the method by which a linear relationship between an observation at time t and the observations at the previous times are measured. The stronger the correlation is in between the output and the previous time series coefficient, the better forecast values will be obtained.

One major question that arises is why can't we use linear regression for forecasting over a time series. Linear regression is a process used to make an estimation of the real values such as the number of customers, total sales, cost of houses, etc. A relationship is developed between a dependent variable represented by Y and an independent variable denoted by X which can be fit in the best line. This best fit line is called as a regression line which can be represented by the following linear equation:

$$Y = a * X + e \tag{2.3}$$

where Y is a vector of observed values, X is a vector of the predictor variables, a is a vector of regression coefficients and, e is the error.

Examples of some of the usage of the autoregressive model include: a model for forecasting the price of crude oil, for finding Internet time delay improving dynamic performances for real time applications, control systems, long term wind prediction that is used for a renewable energy source. One major drawback of the autoregressive model is that past values will not always be the best predictor of the future values specially in the case where the fundamentals of the data are changed.

2.6 Current System Architecture

This section will provide an in-depth discussion of the current vKey client-server architecture, how it is a trusted endpoint, how the deployment takes place, and what components are involved in its operations. The figure and architecture in this section are proprietary of DLS Technology (Ottawa, Canada).

Currently, vKey is a software that is loaded onto any bootable media, for example, a USB stick, hard drive or a MicroSD card. It acts like a deployed virtual laptop but with trusted secure remote access for both thin and thick clients. A thin client is a network computer without a hard disk and acts as a simple terminal to the server. They are generally used where the user has a well-defined and systematic number of tasks for which the system is being used. On the other hand, thick clients perform the bulk of the processing in client-server applications. Thick clients are generally found in nature of operating systems and software, as they can handle thier own resources.

So how does vKey become a trusted endpoint? vKey uses a computer's screen and peripheral devices while bypassing the local hard disk. It leaves a zero footprint of the user information since it does not read, write or store anything from the computer. Once the user plugs in vKey device into a personal computer and reboots it, vKey device instantly turns the personal computer into a secure and trusted computer. It provides industry standard secure virtualization technology that allows users to launch and access enterprise applications and data hosted in a secure cloud computing environment.

Figure 2.12 explains the working model of the current vKey solution. In the external network, there are various users that are connected to the De-Militarized Zone (DMZ) via the Internet. The DMZ is an isolated network provisioned between the internal and the external networks. It is a physical or logical network that links the organization's external-facing services to the Internet. It adds an additional layer of security to the organization's local area network. In DMZ, vDM appliance and vKey help desk are placed.

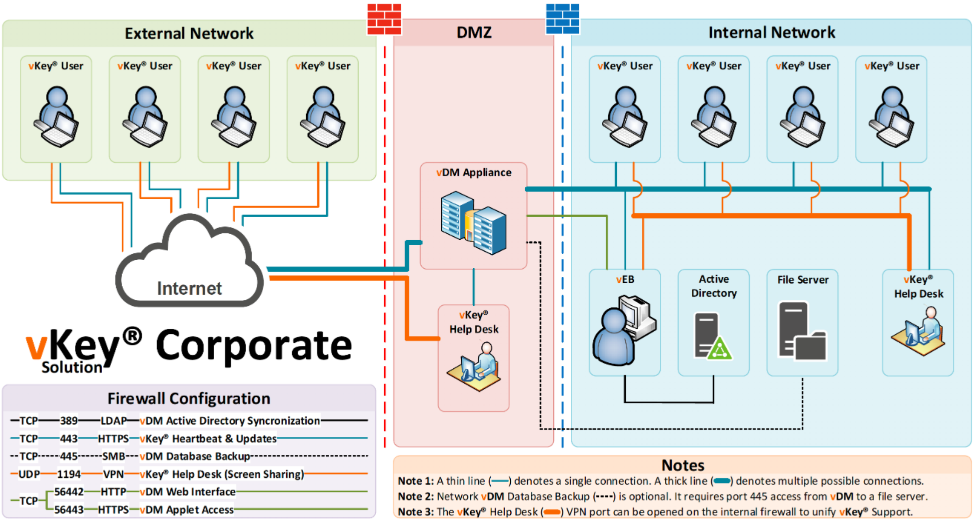

External Network
DMZ
Internal Network
vKey® User
vKey® User
vKey® User
vKey® User
vDM Appliance
vKey® Help Desk
vKey® User
vKey® User
vKey® User
vKey® User
vEB
Active Directory
File Server
vKey® Help Desk
Internet
vKey® Corporate Solution
Firewall Configuration
TCP 389 LDAP vDM Active Directory Syncronization
TCP 443 HTTPS vKey® Heartbeat & Updates
TCP 445 SMB vDM Database Backup
UDP 1194 VPN vKey® Help Desk (Screen Sharing)
TCP 56442 HTTP vDM Web Interface
56443 HTTPS vDM Applet Access
Notes
Note 1: A thin line (—) denotes a single connection. A thick line (▬) denotes multiple possible connections.
Note 2: Network vDM Database Backup (----) is optional. It requires port 445 access from vDM to a file server.
Note 3: The vKey® Help Desk (▬) VPN port can be opened on the internal firewall to unify vKey® Support.

Figure 2.12: Current vKey architecture

The vKey Device Manager (vDM) utility is a management service that allows an administrator to manage, monitor, and configure vKey virtual machines, including the ability to capture an image from the device and building a physical device from a catalog of vKey images which is later distributed to clients. In the internal network, the active directory module acts as a single point of access for system admin in a centralized system which is responsible for handling the user data, security, and resource distribution. The file server is responsible for file storage and transporting the files using the file transfer protocol over the network. vEB is a physical USB device that is used to create other physical vKey software.

2.7 Methodologies to Allocate Virtual Machines in the Cloud

This section discusses the state of the art approaches for achieving the goal of deploying virtual machines in a cloud environment. To begin with, A. Beloglazov *et al.* [18] say that with the introduction of cloud computing, data centers have become much more power efficient since they can run multiple virtual machines on a single physical server. This allows servers that are not running virtual machines to be turned off or put to sleep thus consuming less power. However, the downside of reducing the number of physical servers leads to a reduction in performance. Hence, to address the power-performance trade off, the authors propose an energy-efficient VM placement algorithm which aims to reduce the power consumption of data centers. They propose to exploit all the VM copies which increase the average utilization of the servers. To place copies of the VM without affecting the energy efficiency, the algorithm makes use of the dynamic programming to determine the number of VMs which can be placed on the servers. To improve the efficiency of the algorithm, the paper uses a local search algorithm to examine the total energy consumption in the system. If the utilization of a given server is less than the threshold set by the cloud provider, it will shut down. If the total power consumption is found to be less than the power consumption from the previous configuration, the new configuration is saved. The paper aims to reduce the energy costs by 20%, but it fails to consider the processing load when a VM is being copied from one server to others. The paper also fails to discuss what will happen to the virtual machines that might be running on the server when the algorithm decides to turn it off based on the utilization threshold. The decision to determine the VM placement ignores the communication resources and secondary storages. Also, this paper fails to address the possibility of VM failures while it is being copied.

A survey by Z. Mann *et al.* [19] discuss how cloud computing had overcome the storage problem of a large sum of data using features such as unlimited scalability, and pay per use services to users. The virtual machine allocation is determined by thier placement on the host and how virtual migration plays a role in the energy consumption. The major problem discussed in this paper for the allocation of virtual machines is energy minimization. Energy minimization has always been a concern for cloud providers. Resource management is another main concern for building an efficient and profitable cloud data centers. Proper management will lead to the proper number of servers being active (others will be turned off or sleeping). The researcher′s approach for the energy consumption is based on the virtual machine migration from one server to another. This approach highlights a solution to the problem discussed in [18] about what will happen to the virtual machines running on the server. However, this paper fails to address the issue of allocating the virtual machines depending on the client workload.

Further, M. Nejad *et al.* [20] describe a novel auction-based technique to solve the problem of allocating the virtual machines to the users. The problem is handled by providing the resources either by static provisioning or by dynamic provisioning. In static provisioning, the cloud provider provides the resources to the users before even knowing their demand. Whereas, in dynamic provisioning, resources are allocated based on the user request. Dynamic provisioning has proven to be more efficient and thus, the market demands are fulfilled efficiently.

Later, S. Zaman *et al.* [21] talk about how cloud computing providers use fixed allocation mechanism for allocating virtual machines to their users. The paper reveals that the combinatorial auction-based mechanism can significantly improve the allocation efficiency while generating higher revenue for the cloud providers. In the paper, they proposed two methods of a combinatorial auction-based mechanism for solving the allocation problem of virtual machines in a cloud environment. But in both of the proposed methods, the authors fail to talk about the effect of the load of the clients arriving in the system, as clients have a dynamic demand for virtual machines.

Gagliano *et al.* [22], further advance the research done in [21] by investigating the effect of allocating the resources required for computations via auctions, in which the client is intelligently able to calculate the required resources. But, the calculation done by the client could lead to increased idle resources which are not the best solution in terms of server utilization.

Further, Gomoluch *et al.* [23] proposed a double auction protocol for allocating virtual machines in a cloud and proved that it is better than the conventional approaches discussed above. Later, Lehmann *et al.* [24] proposed a study for combinatorial auctions for single-minded clients and designed a greedy mechanism for the auction, which was used by [19] to allocate virtual machines in the cloud.

Lastly, the effect of allocating virtual machines using these auctions-based techniques on the commodity market is discussed by the paper proposed by Wolski *et al.* [25]. It shows the relation between the price and the market stability that could help the provider to estimate its monetary values.

From the research papers discussed above, we got the idea of making a system that will dynamically allocate virtual machines to users in the cloud. However, the question to answer is how to make this dynamic nature possible. To do so, we further looked into researches that focus on using prediction-based techniques to analyze the client workload. These are described below.

In [26] Zhu *et al.* introduced a method to vertically scale the configuration of the resources such as VM type, VM memory and VM storage. Vertical scaling is the process in which the number of virtual machines remains the same but the amount of resources allocated to them is increased based on the demand. Also, they used an ARMAX method to forecast the CPU cycles and the memory configurations that would be required for successfully hosting an application. This model is best to use if the information on the resources required is decided by the users.

A different model, proposed by Bonvin *et al.* [27] scale the servers based on the past performance and the profit made. In this model, the number of virtual machines can either be increased or decreased and the number of resources allocated to the virtual machines can also be increased depending upon the requirements. However, this model does these changes on the entire host and therefore might lead to wastage of resources.

The systems discussed above are reacting to the changes made in the client workload but only after evaluating the utilization and the throughput of the system. Therefore, if the changes in the client arrival behavior are quicker than the modification time, then clients will experience a delay in getting the virtual machines unless extra virtual machines are available. Assuming that the changes in the client workload are dependent on the time, a time-series prediction model could be able to solve the above problem. It could proactively reconfigure the system before the demand for virtual machines increases greatly. This way, the systemwould be ready to accommodate the incoming clients.

Next, Tang *et al.* [28] introduce a novel algorithm to deploy virtual machines in a cloud computing environment. The proposed model uses a bin-packing algorithm to place the virtual machines. Also, the paper uses the concept of forecasting theory to predict the order of the autoregressive model via the least square method. The coefficients obtained using this method are combined with the bin-packing algorithm and as a result, the number of virtual machines decreases. We will be using a similar idea to reduce the number of virtual machines but with a different approach.

Further, Shaw and Singh [29] designed a double exponential smoothing model to make a decision on live migration of running virtual machines and also decide the appropriate host for migration. In order to reduce the number of VM migrations, the paper proposes an algorithm that decides the importance of migrating a virtual machine based on the present and future workload. To make a prediction about the future workload, the paper uses an exponentially smoothing technique, where all the data from the past are given equal weightage. As the user's behavior is dynamic in nature, the paper fails to address a possibility where we might have to give higher weightage to some specific values in the past data. Also, the paper does not propose any method to select the VM for migration.

A widely used technique named linear regression is able to make faster predictions compared to autoregression. However, it demands a simple data set in comparison to time-series. In papers [30] and [31], it is proven that the workload in data centers shows a behavior that could easily be addressed by the time series model. Therefore, it would be appropriate to use autoregression for forecasting the client workload in our proposed adaptive system. A similar approach is introduced by Rodrigo *et al.* [32] for dynamic provisioning of resources for SaaS applications by using the ARIMA model to analyze workload.

In conclusion, various papers have proposed different algorithms to allocate virtual machines in a cloud environment to reduce the clients waiting time. Also, various schemes are discussed to provision the resources to a virtual machine with the aim to reduce the number of un-utilized resources. However, all the studies discussed above fail to address the issue of allocating the virtual machines depending upon the load of the clients arriving in the system. Further, the existing state of the art systems does not address the issue of the client's data privacy. Therefore, to address these issues, a new prediction-based technique to allocate virtual machines in a cloud environment is described in the next chapter.

Chapter 3

System Design and Implementation

In this chapter, three models are proposed for allocating virtual machines in the cloud. These virtual machines have a pre-installed secure remote access platform called vKey (discussed in Section 2.6) which is used for secure login. The first model, referred to as Baseline System (BS), uses a traditional approach of allocating virtual machines to users. Upon receiving a request, this model creates a virtual machine from a pre-installed template and allocates it to the user. In the second model, named Reactive System (RS), a fixed number of virtual machines are pre-created and kept in a halted state in the Citrix XenServer. When a client demands a virtual machine, one of the virtual machines is switched to the running state and allocated to the client. The third model, reffered to as Proactive System (PS), is the model that has been delivered to DLS Technology. PS is an adaptive learning system in which the number of virtual machines that are created a-priori is dynamic, unlike RS. In this model, the system continuously collects the arrival data to learn the behavior of the clients and uses it to predict the number of virtual machines that need to be created a-priori in a certain time period. The rest of this chapter discusses the three proposed systems in detail. More precisely, Section 3.1 explains the baseline system and the algorithms associated with it. This section also explains the algorithms and working of the modules which are common across the three systems.

Next, Section 3.2 and Section 3.3 explain the new modules that have been added for the reactive and the proactive systems respectively.

3.1 Baseline System

The baseline system uses a very simple and traditional approach for allocating a virtual machine to incoming clients requests. In this approach, no virtual machines are created a-priori. Therefore, when a client requests for service, a virtual machine is copied from a pre-defined template and then allocated.

As shown in Figure 3.1, clients are an external component of the system. The baseline system consists of an allocator which further consists of three daemons running in parallel that are connected through TCP sockets. The various components of the baseline system are explained below.

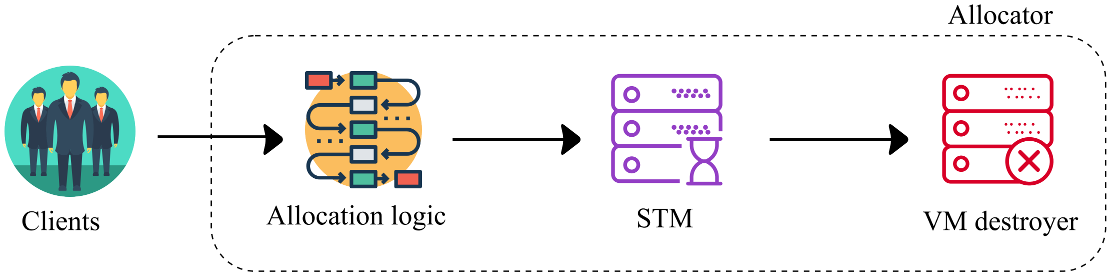


Figure 3.1: Architecture of the baseline system

- Clients: Multiple clients arrive independently of one another and request a VM for a particular period of time. Clients do not always request the same type of VM. For example, a client from company A will have access to a different VM than a client from company B. To indicate which type if VM is requested, we used the following notation VM_x. For simplicity and without loss of generality, the systems proposed in this thesis has been developed and tested with 2 types of services: VM_A and VM_B.

The allocator is the key component of the baseline system which is composed of three daemons as discussed below:

- Allocation logic: The allocation logic is the brain of the allocator. It receives the client requests consisting of the Client ID *($ClientID^i$)*, Arrival Time *(A_t^i)*, Service time *(H_t^i)* - time for which a virtual machine is used by the client and type of the VM requested *(VM_x^i)* by the i^{th} client. Once the request is received, it gets queued in a First In First Out (FIFO) queue. Then, the first request from the queue is picked up and starts getting processed. Depending upon the type of request, the capacity requirements of the current host server are checked and if these constraints are satisfied, a new VM is created and allocated to the client. If there is not enough space in the current host server, the allocator logic checks whether another host server is available in the cluster. If the host is available, then the same process is repeated unless there is no host available. When no host is available in the cluster, the client request is rejected.

- Service Time Manager (STM): The responsibility of the STM is to keep track of the service time for all the currently active clients. The reason for making it a separate daemon is that, as we are simulating the client requests and hence when a virtual machine is started for a client, we need to decrement virtual machine hold time. For decrementing the VM hold time for each client, the current thread needs to go to sleep for one second. Hence, the STM is a multi-threaded daemon where one thread keeps track of the VM hold time and another thread receives the VM hold time requests from the allocation logic and appends them to a FIFO queue. Once H_t^i reaches zero for a particular client, the Client ID and UUID are passed to the VM destroyer.

- VM destroyer: The job of the VM destroyer is to destroy virtual machines. However, before destroying a VM, it has to be in the halted state and also the data associated with the client needs to be deleted. It does the following tasks in the given order.
  - Stop virtual machine: After the service of the client is completed, the VM is halted.
  - Delete virtual disk: It means deleting all client specific data stored in the default storage repository and memory.
  - Delete virtual machine: Finally, the VM is being destroyed from the Citrix XenServer.

For each request, the timestamps are collected and added to the database for performance analysis. Figure 3.2 shows the flowchart of the baseline system and describes how a client request is processed. The algorithm number (written as 'Algo 19' in short) in the flowchart represents the algorithm used for the respective module. Note that the same convention is used in the other flowcharts (Figure 3.6 and Figure 3.11).

3.1.1 Initial Connection Setup of the Baseline System

The initial connection setup shown in Figure 3.3 needs to be done for the baseline system. It is imperative to mention that this sequence diagram only shows the positive

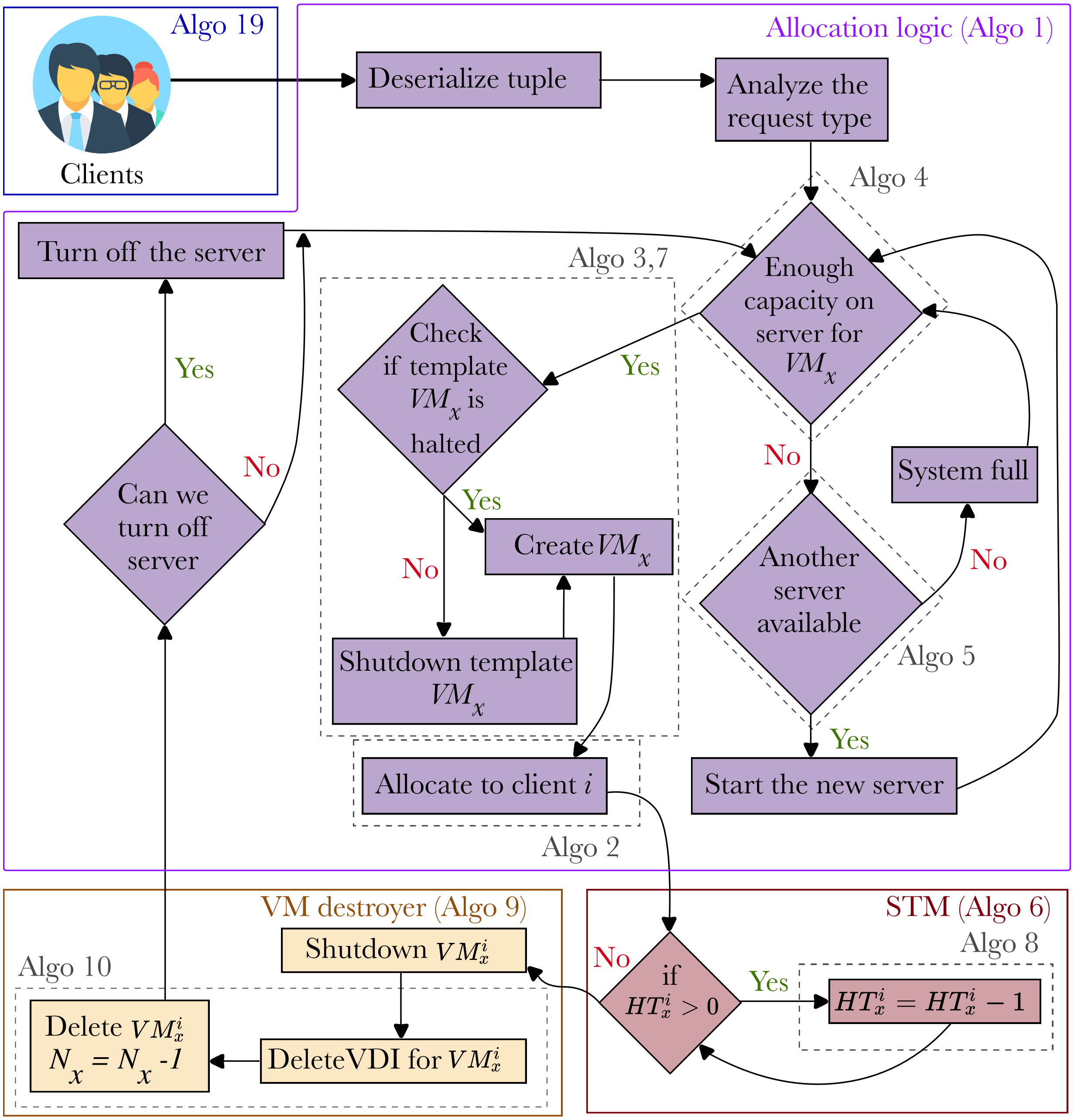


Figure 3.2: Flowchart for the baseline system

cases which mean that the connection is made when the request is sent. This is done for sake of simplicity but in case the connection request is rejected, it needs to be sent again. Initially, the Citrix XenServer needs to be turned on, so that it can receive requests. The different modules are connected to one another via a client-server model, therefore the socket server needs to be started first so that it can receive a request from a client socket. After the Citrix XenServer is turned on, the VM destroyer sends a connection request to the Citrix XenServer. If these connections are successful, then a socket server is turned on at port x. Further, the service time manager daemon will send a connection request to the Citrix XenServer. Once, the connection is successful, a socket server will be started on port y. Similarly, the allocation logic establishes a connection with the Citrix XenServer, which in-turn starts a socket server on port z. As all socket servers are up and running, now the STM will send a client socket request from port x to port y and establishes a client-server connection between STM and VM destroyer. Similarly, a client-server connection is established between the STM and the allocation logic using ports y and z respectively. The order in which the various daemons need to be started within the allocator.

$$\text{VM destroyer} \rightarrow \text{STM} \rightarrow \text{Allocator logic}$$

Once all the connections are successfully made, the allocator is ready to handle the client requests and waits for them.

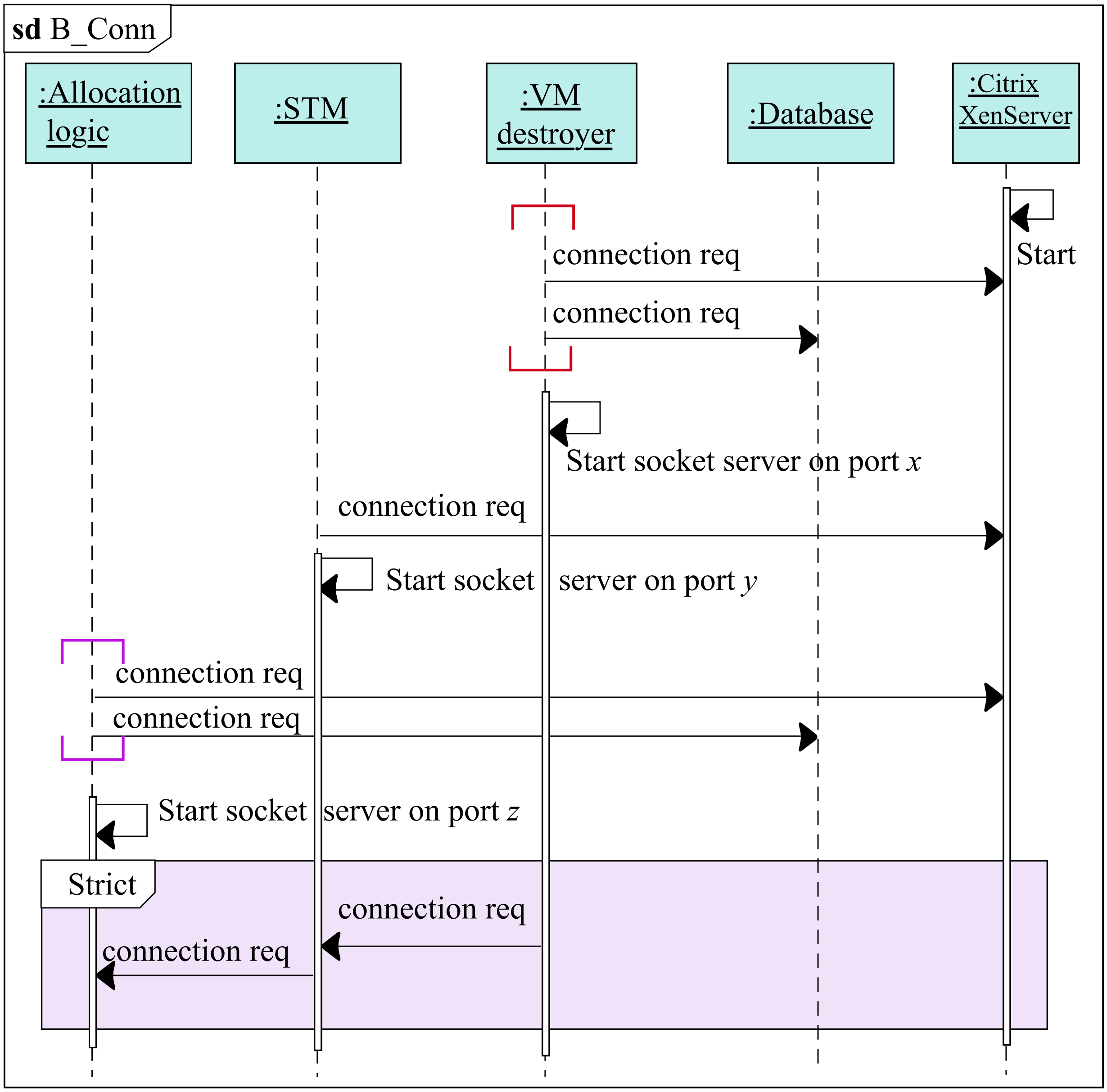


Figure 3.3: Initial connection setup for the baseline system

3.1.2 Allocation Logic

The allocation logic is the brain of the allocator. It receives the client requests and according to the VM required by the client, that particular type of virtual machine is created from a given template and then allocated to the client.

As shown in Algorithm 3.1, initially, the allocator logic daemon tries to login to the XenServer using a valid username and password. If the login is successful, it initializes the server socket object which is responsible for handling the client service requests. Following this, a client socket object is initialized which sends the time management requests to the STM (here we assume that the STM is up and running to receive time requests from the allocation logic). In case the login is not successful, an error is shown on the console (Line 21). On a successful connection, the allocator starts accepting client requests. Requests coming from the clients are de-serialized and then parsed to get $ClientID^i$, H_t^i and VM_x^i where i denotes the i^{th} client and, x denotes the type of the virtual machine requested by the client. These service request tuples are appended to a FIFO queue and served in a first-in-first-out manner. Then, every request is sent to the next method called *CreateAndAllocateVM*. The role of *CreateAndAllocateVM* is to serve a client request by creating the required type of VM and then allocating it to the respective client followed by starting of the virtual machine. Also, time stamping is done as a client arrives at allocator logic.

Algorithm 3.2 is a pseudo-code to understand the process of creating a virtual machine of type x as per the client demands. The method *CreateAndAllocateVM* further calls another method called *CreateSingleVM*. This method creates a virtual machine by tak-

Algorithm 3.1: BaselineAllocatorLogic

```
Input: Login credentials for XenServer, Service Time manager and VM Destroyer
       need to be up and running
Output: VM creation request sent to CreateAndAllocateVm and other clients are
        waiting in queue
Try to Login into XenServer                                    ▷ using valid credentials
if connection with XenServer ← True then
    Create server socket object to receive client requests
    Create client socket object to send connection request to STM
    if Pool logic is connected to STM then
        while Pool logic and STM connection is persistent do
            Accept client connection request
            while Client service requests ← True do
                Get $ClientID^i$                        ▷ De-serializing service request tuple
                Get $H_T^i$
                Get $VM_x^i$
                Get $A_T^i$
                Enqueue service request tuple to FIFO queue
                foreach  earliest service request ∈ queue do
                    Call CreateAndAllocateVM(RequestTuple) method
                end
            end
        end
    end
else
    Handle exception and show error message
end
```

Algorithm 3.2: CreateAndAllocateVM

```
Input: $ClientID^i$, $H^i_T$, $VM^i_x$, $A^i_T$, $Session_{id}$
Output: VM created successfully if requirements are met OR VM not created
isCreated ← CreateSingleVM($Session_{id}$, $VM^i_x$)
if isCreated ← True then
    Get $VM_{CreationStartTime}$
    Get $VM_{CreationEndTime}$
    Get $UUID$ of VM which is created                         ▷ This is unique ∀ VM’s
    Allocate VM to client
    Dequeue current request
    Start virtual machine
    Get $VM_{StartTime}$ ← now()                              ▷ System time in milliseconds
    Generate JSON tuple
    Serialize tuple[$UUID$, $H^i_T$]                          ▷ using pickle.dumps()
    Send request tuple to Service Time Manager using client socket
    Write $ClientID^i$, $VM_{CreationStartTime}$, $VM_{CreationEndTime}$, $VM_{StartTime}$, $UUID$ to DB
    Commit to DB
else
    Reject service request          ▷ As VM memory or Storage requirements are not met
    Dequeue request from FIFO Queue
    Set $VM_{StartTime}$ ← null
    Write $ClientID^i$, $VM_{CreattionStartTime}$, $VM_{CreationEndTime}$, $VM_{StartTime}$, $UUID$ to
      DB
end
```

ing $Session_{id}$ and VM_x^i as input. Here, $Session_{id}$ refers to the unique session ID which is provided by XenServer. Based on the client request type (VM_x^i), *CreateSingleVM* method creates a virtual machine of type x if all the constraints discussed in Section 3.1 are satisfied. Further, this method returns true, if a VM is created and false otherwise. This value is saved to a boolean variable named *isCreated*. If the virtual machine is created (lines 2-14) then various data points are retrieved and saved such as $VM_{CreationStartTime}$, $VM_{CreationEndTime}$ and *UUID*. $VM_{CreationStartTime}$ represents the time when the virtual machine creation process was started. $VM_{CreationEndTime}$ is the time stamp at which the creation process of a VM is completed and *UUID* is the unique reference given by the Ctirix XenServer for each virtual machine it creates. This reference is very important as it is required to perform various operations like halting, starting, restarting or destroying a virtual machine once the user ends a session. Further, this virtual machine is allocated to the client and the service is started. As soon as the service is started, the service time is noted and saved to a variable named $VM_{StartTime}$. It is important to mention that the tuple ¡ClientID, UUID¿ represents which virtual machine is allocated to which client. As soon as the service is started, a JSON time request tuple is generated, serialized and sent to the next daemon called Service Time Manager. The starting of the virtual machine and sending the time tuple to the Service Time Manager happens in parallel. Furthermore, this data is written to a database and committed. If the virtual machine is not created, then the client request is rejected and dequeued from queue followed by setting values to null in the database.

Algorithm 3.3 explains how a virtual machine is created based on VM_x^i where $x \in$ A or B. Before starting the creation of a virtual machine, the $VM_{CreationStartTime}$ is set to the

Algorithm 3.3: CreateSingleVM

```
Input: $Session_{id}, VM_x^i$
Output: $isCreated, UUID, VM_x, VM_{CreationStartTime}, VM_{CreationEndTime}$
Set $VM_{CreationStartTime}$ ← now()
Select $VMTemplate_x$ based on requested $VM_x$
areRequirementsMet ← CheckVMConstraints($Session_{id}, VMTemplate_x$)
if areRequirementsMet ← true then
    vms ← session.xenapi.VM.getAllRecords()
    foreach vmRef ∈ vms do
        Get $vmRec$ using $vmRef$          ▷ Reference will by used to copy template
        if $vmRec[\text{'}state\text{'}]$ ← halted & vmRec['nameLabel'] ← $VMTemplate_x$ then
            Get $VMTemplate_{ref}$
            break
        end
    end
    SelectedStorageRepo ← defaultStorageRepo          ▷ hard-disk for ex.
    Perform network settings
    Create $VIF$ instance(Device, Network, VM, MAC, QoS Type)
    $CreatedVMRef$ ← session.xenapi.VM.copy($VMTemplate_{ref}, vmName,$
      $SelectedStorageRepo$)
    Get $UUID$ of created VM using $CreatedVMRef$          ▷ for future reference
    Add kernel command line          ▷ Can be interactive ∨ non-interactive
    Set $VM_{CreationEndTime}$ ← now()
    Set $isCreated$ ← True
    Return $isCreated, UUID, VM_x, VM_{CreationStartTime}, VM_{CreationEndTime}$
else
    Set $isCreated$ ← $False$
    Set $UUID$ ← $null$
    Set $VMCreation_{StartTime} \land VMCreation_{EndTime}$ ← $null$
    Return $isCreated, UUID, VM_{type}, VM_{CreationStartTime}, VM_{CreationEndTime}$
end
```

current system time in milliseconds. It is done so that we can find out how much time the VM creation takes. Then, based on the type of service requested by the client, that type of VM template is selected. If service of type A is requested, then $VMTemplate_A$ is selected else $VMTemplate_B$ is selected. This method further makes a call to the *CheckVMConstraints* method to ensure that all the constrains are satisfied. The rationale behind this is that, before creating a virtual machine, we need to ensure that the minimum requirements of the host such as memory and storage are met so that client service can run seamlessly. If the *CheckVMConstraints* method returns true, it means that all constrains are satisfied and there is enough space and memory in the host to accommodate the request. If the *CheckVMConstraints* method returns false on Line 23, it means that all the hosts are either out of memory or storage, and the client is rejected with the message "Try again later". In the case where the requirements are met, we need to make sure that the template from which we want to copy is in the halted state (as shown in lines 4-21). This is done by iterating through all the templates of the virtual machines and selecting the required template VM. Next, we need to select the storage that we will be using for the virtual machine. In the Citrix XenServer, we can make a particular storage as our default storage repository. After selecting an appropriate storage, we need to configure the network. Further, an *VIF* instance is created by providing various parameters as mentioned at Line 15. Finally, the virtual machine is copied by providing template reference and storage to the *session.xenapi.VM.copy* method provided by the Citrix XenServer. We can also set the name of the virtual machine to be created in this method. After all these configurations are done, the $VM_{CreationEndTime}$ is timestamped according to the system's current time.

Algorithm 3.4: CheckVMConstraints

```
Input: $Session_{id}, VMTemplate_x, MemoryThreshold_x, StorageThreshold_x$
Output: reRequirementsMet ← true or false
hostList ← session.xenapi.host.getAll()                      ▷ Getting all hosts in cluster
srList ← session.xenapi.SR.getAll()                          ▷ Getting all storage repositories
foreach currentSr ∈ srList do
    srName ← session.xenapi.SR.getNameLabel(currentSr)
    if srName ← “hd” then
        Total storage ← session.xenapi.SR.getPhysicalSize(currentSr)    ▷ In bytes
        Used storage ← session.xenapi.SR.getPhysicalUtilisation(currentSr)
        Get Total memory and Used memory in bytes
        Make unit conversion if required                     ▷ From bytes to GB
        FreeStorage ← (Total storage - Used storage )
        FreeMemory ← (Total memory - Used memory )
    end
end
if $VMTemplate_x$ ← $VMTemplate_A$ then
    if FreeStorage > $StorageThreshold_A$ & FreeMemory > $MemoryThreshold_A$
     then
        return areRequirementsMet ← true
    end
    else
        Call CheckAnotherHost(hostList, AllVariablesReceivedByCurrentFunction)
    end
end
else if $VMTemplate_x$ ← $VMTemplate_B$ then
    if FreeStorage > $StorageThreshold_B$ & FreeMemory > $MemoryThreshold_B$ then
        return areRequirementsMet ← true
    end
    else
        Call CheckAnotherHost(hostList, AllVariablesReceivedByCurrentFunction)
    end
end
```

Algorithm 3.4 shows the pseudo-code for the *CheckVMConstraints* algorithm. This algorithm is responsible for checking whether the memory and storage constraints for a particular type of virtual machine are satisfied or not. This method initially checks the requirements of the current host and if for some reason, the current host is not able to create a virtual machine, then requirements are checked for all other hosts in the cluster unless there in no host available. *CheckVMConstraint*s requires the least memory and storage requirements denoted by $MemoryThreshold_x$ and $StorageThreshold_x$ respectively, where x represents the type of virtual machine. This method also requires the knowledge about which template to use for creating a respective type of virtual machine. Initially, a list of all the available hosts in the cluster and all the storage repositories are obtained using the *session.xenapi.host.getAll()* method which is provided by the Citrix XenServer API. These lists are then stored in the variables *hostList* and *srList* respectively. The *srList* is iterated until we find a storage type in which we want to create a virtual machine, which is a hard disk in our case. After selecting "hd" as our preferable storage type, we get the total storage and the used storage in bytes using *getPhysicalSize* and *getPhysicalUtilisation* methods. In a similar way, the total memory and used memory are computed for the current host. As the XAPI returns the values in bytes, we can make the necessary conversation to MegaByte (MB) or GigaByte (GB). Free storage and free memory are computed by subtracting the used value from the total value. Once this is done, the type of the template is then checked form which we need to copy. If the selected template is $VMTemplate_A$, then memory and storage conditions are checked as shown in Line 14. If both of these constraints are satisfied, then *areRequirementsMet* variable is set as true. If aforementioned conditions are not satisfied, then *CheckAnotherHost* method is called which will check if these requirements are satisfied in another machine. *CheckAnotherHost* will recursively check for all available hosts in a cluster. As shown in Line 22, if the selected template is $VMTemplate_B$, then constraints for memory and storage are checked and the same steps are repeated.

Algorithm 3.5: CheckAnotherHost

```
   Input: $hostList$, $Session_{id}$, $VMTemplate_x$, $MemoryThreshold_x$, $StorageThreshold_x$
   Output: isAvailable ← true or false
1  numberOfPeers ← session.xenapi.host.getHaNetworkPeers(hostList)
2  if numberOfPeers ← 0 then
3      return $isAvailable$ ← false
4  end
5  else if numberOfPeers $\geq 1$ then
6      foreach $host \in hostList$ do
7          call $CheckVMConstraints(parameterlist[...])$        ▷ recursive calling
8          return $isAvailable$ ← true
9      end
10 end
```

Algorithm 3.5 is a method to check whether another host is available or not. First, it finds the total number of hosts in the cluster by using a getHaNetworkPeers() method of XAPI and saves it in a variable called *numberOfPeers*. As shown in Line 2, if there is no other host available in the cluster, then *isAvailable* is returned as false. If numberOfPeers ≥ 1, then *CheckVMConstraints* is called again with the required parameters such as $Session_{id}$, $VMTemplate_x$, $MemoryThreshold_x$ and $StorageThreshold_x$.

3.1.3 Service Time Manager

In the previous section, we discussed how a VM is created and allocated to the client, and the various methods used to verify that all constraints are met. This section will provide insights on what happens when a VM service is turned on or in other words, when clients start using the virtual machine. Figure 3.4 shows how the STM handles all the requests arriving from the allocation logic. There are two threads running in parallel inside the STM: Thread 1 is responsible for accepting the service time requests sent by the allocation logic. These requests are then de-serialized, parsed and appended to a shared *ClientsToBeServed* list. Thread 2 iterates over the *ClientsToBeServed* list and checks whether the service has finished or not. It does it by checking if the remaining service time is > 0. If the service is finished, then Thread 2 appends the particular finished request to a *ClientsFinishedServing* list. Once, Thread 2 has finished an iteration on the entire list, it sleeps for one second using *thread.sleep(1)* and then decrements the remaining service time of all the clients in the *ClientsToBeServed* list by 1 second. Next, the data in the *ClientsFinishedServing* list is passed to the VM Destroyer such that the list can be reused for the next iteration. Please note that one of the major reasons for having two separate threads in the STM is that in order to decrement time, the incumbent thread has to sleep and hence it will not be able to receive the time requests sent by the allocation logic. An explanation of the STM daemon is given below with the help of pseudo-code.

Algorithm 3.6 explains the working of the STM daemon. Before starting this daemon, we need to ensure that the VM destroyer is up and running so that it can receive requests from the STM. After successful login to the Citrix XenServer, a connection request is sent to the VM destroyer. If this request is accepted, then two threads start in parallel. These

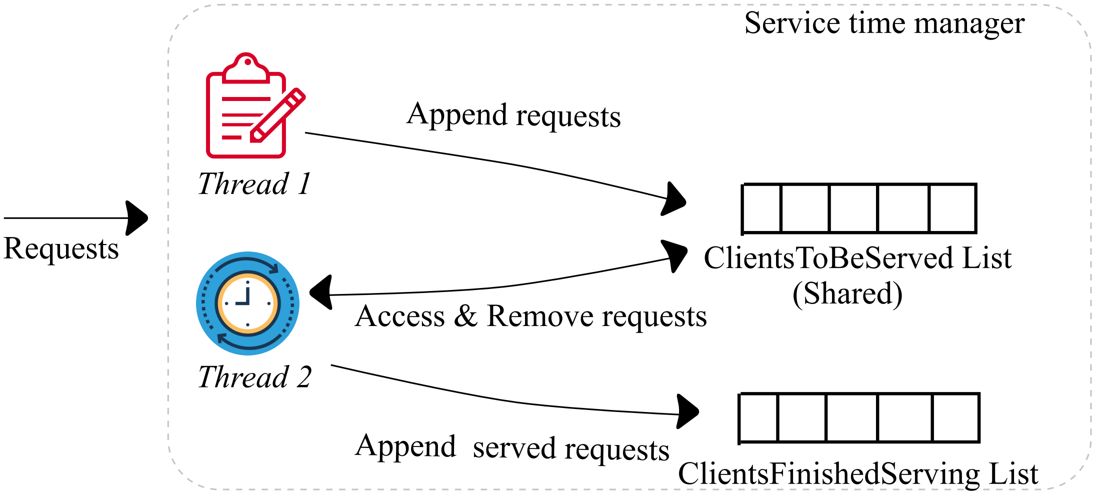


Figure 3.4: Overview of the service time manager

```
Algorithm 3.6: ServiceTimeManager
  Input: Login Credentials for XenServer, VM Destroyer is up and running
  Output: Stops the requests after service is over and writes to database
  Create server socket object and wait of connection from Pool Manager
  Create client socket object to send stop requests to VM Destroyer
  Try to login to XenServer
  if Login to XenServer ← successful then
      Send connection request to VM Destroyer
      if Service time manager & VM Destroyer ← connected then
          Start RequestHandler()                              ▷ on Thread1
          Start Timer()                                       ▷ on Thread2
      end
  end
```

Algorithm 3.7: RequestHandler

```
   Input: Server socket object
   Output: isAvailable
 1 Initialize ClientsToBeServed list
 2 while true do
 3     running ← true
 4     while running do
 5         BinaryTuple ← Receive tuple from Pool Manager
 6         JSONTuple ← Binary tuple          ▷ De-serialize using pickle.loads()
 7         if JSONTuple ≠ null then
 8             Get $ClientID^i, UUID, Service_{time}$
 9             Append $ClientID^i, UUID, Service_{time}$ to ClientsToBeServed list
10         end
11         else if Exception is raised then
12             Running ← false
13         end
14     end
15 end
```

threads are discussed in more detail below.

Whenever the service for a particular client is started, the allocation logic sends a request to the STM. In Algorithm 3.7, the request handler is responsible for receiving the time requests from the allocation logic. The *While* loop in Line 2 is used so that the request handler keeps on receiving client requests until this process is killed explicitly. Binary request tuples are received from the allocation logic and further de-serialized and parsed to form JSON tuples. An *if* condition on Line 7 checks whether the tuple is not null. In case the data format is not valid, an exception will be raised which can be handled appropriately. The received request is appended to a shared *ClientsToBeServed* list for further processing.

Algorithm 3.8 explains the concept of having a time module. This module is responsible for decrementing the service time for various clients. First, the *ClientsToBeServed* list is initialized. A *while* loop on Line 3 makes sure that the list is iterated as long as the system is up and running. Further, an *if* condition on Line 4 checks whether there is a request which is in the *ClientsToBeServed* list. Then for all such requests, the service time is decremented by 1 second. Another *if* condition in Line 7 checks if the service time of that particular client is 0. In this case, the service end time of the i^{th} client is noted by using the system millisecond time and saved to the H_t^i variable. As we are currently iterating over the *ClientsToBeServed* list, when we find that the service for a particular client has been finished, we save it to the *ClientsFinishedServing* list which will be used once we finish up iterating the *ClientsToBeServed* list. Now, as the service is finished, we need to send a VM destruction request to the VM destroyer. This is done by forming a destruction request tuple as shown in Line 10. This tuple is serialized and then sent to the VM destroyer using the socket connection which was set up earlier. Also, we need to save the values to a database for performance evaluation purposes. Hence, the required values are inserted to the database. Note that, we are making database connection in this thread instead of the main thread because most databases allow to write in the database from the same thread from which the connection object was made. At Line 15, a *for* loop iterates over the *ClientsFinishedServing* list and it removes all the finished requests from *ClientsToBeServed* list. After this loop is over, the *ClientsFinishedServing* list is re-initialized to be used in the next iteration. Finally, a sleep time of one second is used to

Algorithm 3.8: Timer

```
Input: ClientsToBeServed list & ClientsFinishedServing list & Database cursor
Output: Service time is decremented & Served clients are removed & VM
        Deletion request is sent to VM destroyer & Database is updated
Initialize ClientsFinishedServed ← null
Database cursor ← Login to database                          ▷ we used Sqilte3
while true do
    if ClientsToBeServed ≠ null then
        foreach item ∈ ClientsToBeServed list do
            $H_T^i \leftarrow H_T^i - 1$
            if $H_T^i \leftarrow 0$ then
                $Service_{EndTime}^i$ = now()                ▷ System millisecond time
                Append this request to ClientsFinishedServing list
                Generate [$Client_{id}, Service_{EndtTime}, UUID$] tuple
                Serialize and send tuple to VM destroyer
                Write to database and commit
            end
        end
        foreach tuple ∈ ClientsFinishedServing list do
            Remove tuple from ClientsToBeServed list
        end
        ClientsFinishedServing ← null
        time.sleep(1)                                        ▷ sleep for 1 second
    end
end
```

decrement time logically.

3.1.4 VM Destroyer

Until now, we have discussed how a virtual machine is created, steps by which it is allocated and steps taken to manage the service time. Now, we will discuss the concept of how to destroy a virtual machine. Destroying the VM and its VDI is a very important step as it will ensure that the entire data of the client is destroyed and cannot be retrieved. VM destroyer is the module designed to receive requests from the STM and is responsible for the destruction of the virtual machines.

Algorithm 3.9 is the pseudo-code which explains the virtual machine destroyer. After successful login to the database, a server socket object is created so that it can receive requests from the STM. Login to XenServer is done using valid credentials. As shown in Line 20, if the login is not successful, then an exception is handled and an error message is shown on the console. On the other hand, if the login is successful as shown in Line 4, then the server waits for requests from the STM. A *while* loop on Line 6 ensures that the STM has a connection with the VM Destroyer. With a successful connection establishment, the system starts accepting VM destruction requests, de-serializes them and raises an exception if there is any issue with the data format. After making sure that requests have arrived in the proper format, they are appended to a FIFO queue for destruction. This step is required as destroying a virtual machine is a multi-step process that needs to be followed. In other words, VM deletion takes a certain amount of time. It makes complete sense to add them to the queue and then delete them in a FIFO manner. A *for* loop in Line 14 picks up the first deletion request and calls the Shutdown&DestroySingleVM() function for each request. Once this request is completed, it is dequeued from the queue and another request is served. A detailed discussion of the Shutdown&DestroySingleVM() method is provided next.

Algorithm 3.10 shows the pseudo-code for shutting down a single virtual machine followed by the destruction of the VDI and the VM. This method is recursive in nature and thus iterates for all the VM deletion requests. Session ID, UUID, Service End Time and Client ID are passed to this function. This is because, after the successful deletion of the virtual machine, the database is also updated. Now, for shutting down a virtual machine a *UUID* is required. Initially, a reference for each virtual machines is saved to variable *vms* by using the *VM.getAllRecords* method of XAPI. A *for* loop on Line 2 iterates through all the VM references and breaks finally when a virtual machine is deleted. The main point of focus here is that before destroying the virtual machine, we need to find the right virtual machine by comparing the UUIDs. Also, the powerState of the virtual machine is checked, which should be in "running state" at this time. Once the required virtual machine has been found, then it is brought to the halted state using the *VM.hardShutdown(vmRef)* function. After the virtual machine has been stopped, we need to delete the data which was used by the client. Client-specific data is stored in VDI which needs to be deleted prior to deleting the VM. A *for* loop on Line 6 iterates through all the available VBDs and then delete the selected VDI. Once the deletion of the VDI is completed, we can destroy the virtual machine using the VM.destroy(vmRef) function. As soon as the VM is destroyed, its deletion time is saved in the $VM_{DestructionTime}$ variable. Furthermore, various variables like *UUID*, $Service_{EndTime}$, *ClientID* and $VM_{DestructionTime}$ are saved to the database and committed.

Algorithm 3.9: VMDestroyer

```
Input: Valid login credentials for the Citric XenServer & VM deletion requests
       from service time manager
Output: Deserialize the deletion request and append it to a queue
DatabaseCursor ← login to the database                    ▷ VM-Destruction Database
Create server socket for receiving requests from Service time manager
xenapi.loginWithPassword(username, password)
if Login with XenServer ← Successful then
    serversocket.listen()                  ▷ waiting for request from time manager ...
    while true do
        running = true
        clientsocket ← serversocket.accept()    ▷ Accept request from time manager.
        receivedData ← clientsocket.recv()
        if receivedData ≠ null then
            Deserialize and parse deletion requests
            Raise Exception if format if not OK
            Append deletion requests to FIFO Queue
            foreach earliest request ∈ Queue do
                Call $Shutdown\&DestroySingleVM(Session_{id}, UUID,$
                  $Service_{EndTime}, ClientID^{i})$
                Dequeue request
            end
        end
    end
else
    Handle exception and show error message
end
```

Algorithm 3.10: Shutdown&DestroySingleVM

```
Input: $Session_{id}$, $UUID$, $Service_{EndTime}$,$ClientID^{i}$
Output: VM's are halted and then destroyed & VDI's are destroyed & DB is
        updated
$vms$ ← session.xenapi.VM.getAllRecords()
foreach vmRef ∈ vms do
    if vmRec[powerState] ← running or vmRec[uuid] ← $UUID$ then
        session.xenapi.VM.hardShutdown(vmRef)        ▷ Step #1 : shutting down VM
        vbds ← session.xenapi.VM.getVBDs(vmRef)
        foreach vbd ∈ vbds do

        end
        vdiRef ← session.xenapi.VBD.getVDI(vbd)
        Try session.xenapi.VDI.destroy(vdiRef)             ▷ Step #2 : destroy VDI
        if VDI destroyed ← True then
            pass
        else
            Raise exception and close relevant socket connections
            Close XenServer connection
        end
        session.xenapi.VM.destroy(vmRef)                   ▷ Step #3 : destroy VM
        $VM_{DestructionTime}$ ← now()
        Write ($UUID$, $Service_{EndTime}$, $ClientID^{i}$, $VM_{DestructionTime}$ ) to database
        Commit to Database
        Print success message on console
        break
    end
end
```

3.2 Reactive System

In the previous section, we allocated virtual machines in a simple way (create and allocate VMs as clients come). But that approach has some major setbacks which will be discussed in this section and proved by the performance evaluation in the next chapter.

In this section, we will walk through the architecture of the reactive system which is pictorially shown in Figure 3.5. Further, we will discuss the new methods we introduced to overcome the drawbacks of the baseline system. To do so, we introduced an idea of maintaining a pool of virtual machines with a fixed threshold or pool size. This way, when a client arrives, there is a virtual machine waiting for it in the pool (this is the goal we are trying to achieve). Thresholds of the virtual machine that are maintained in the pool are defined below:

$$T_A = \textit{Threshold number for type A } (VM_A) \textit{ machines} \tag{3.1}$$

$$T_B = \textit{Threshold number for type B } (VM_B) \textit{ machines} \tag{3.2}$$

When comparing Figure 3.5 to Figure 3.1, it is easy to observe that another daemon called VM creator has been added to the architecture. The reason to do so will be clear as we dig deeper into the reactive system. Also, we have introduced the pool manager which has the role to maintain a smooth flow of data from one daemon to another and also allocate the virtual machines to clients similar to allocator in Figure 3.1.

Also, in this section, we will explain only those daemons and algorithms which are different or new from the baseline system explained in Section 3.1.

1. Clients: This module works the same way as we have discussed in Section 3.1.

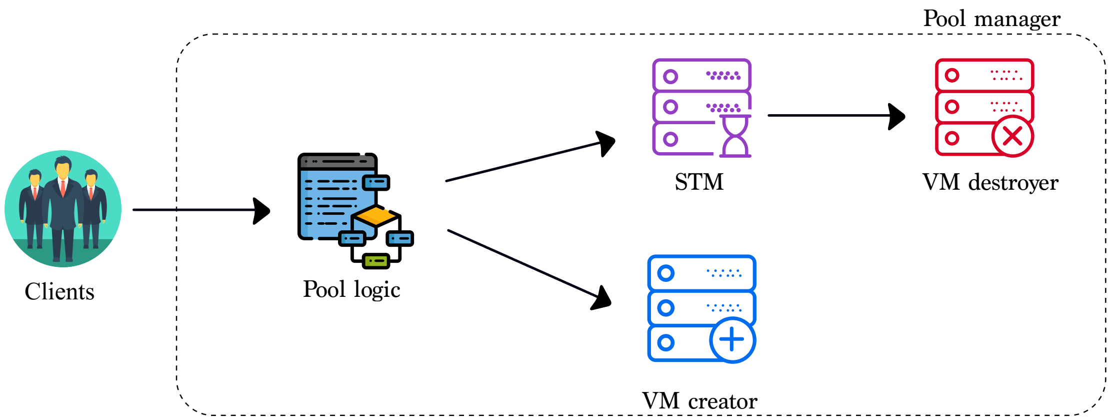


Figure 3.5: Architecture of the reactive system

Multiple clients arrive asking for the service of a particular type.

2. Pool logic: This module is more advanced and different from the allocation logic of the baseline system. It is the brain of the pool manager. It receives the client request which has the information about the Client ID, Arrival Time, Service Type as discussed in Section 3.1. When a request arrives, it is queued in a FIFO queue. Then, the pool logic picks the first request in the queue and checks the type of the virtual machine that has been requested and then checks whether that particular type of virtual machine is present in the pool of virtual machines or not. If the virtual machine is present, it is allocated to the client and in parallel on a second thread, a similar type of virtual machine is created from the template and placed in the pool in order to maintain a constant pool size. If the virtual machine is not present in the pool, a request is sent to VM creator with the information of which type of virtual machine needs to be created while the client waits in the queue. Once, the creation process is over, the virtual machine is started and allocated to the client. Please note that the pool logic is only allocating the virtual machine but

not creating them. This makes it different from the allocator logic studied in the previous section.

3. Service Time Manager: The STM works the same way and has the same responsibilities as discussed in the baseline system (see Section 3.1.3).

4. VM creator: In this module, there are two threads which work in parallel. The first thread, is responsible for receiving the requests coming from the pool logic and appending them to a FIFO queue. Then, the second thread picks the first request from the queue and checks the type of virtual machine that needs to be created. Upon identifying the type, it starts to create a virtual machine and then places the created virtual machine back to the pool.

5. VM destroyer: This daemon is responsible for deleting the virtual machine and its disk image. The process of how it is done is explained in Section 3.1.4.

Figure 3.6 shows the flowchart for a clients request in the reactive system and how are different algorithms linked to each other and are marked with the corresponding algorithm number.

3.2.1 Initial Connection Setup for the Reactive System

Figure 3.7 explains the initial connection setup of the pool manager of the Reactive System. The first setup is to turn ON the Citrix XenServer so that it can accept all the incoming requests from the pool manager. As discussed, our architecture is based on the client-server socket and therefore, we need to ensure that the server socket is running smoothly in order to receive client socket requests. Once, the Citrix XenServer is up

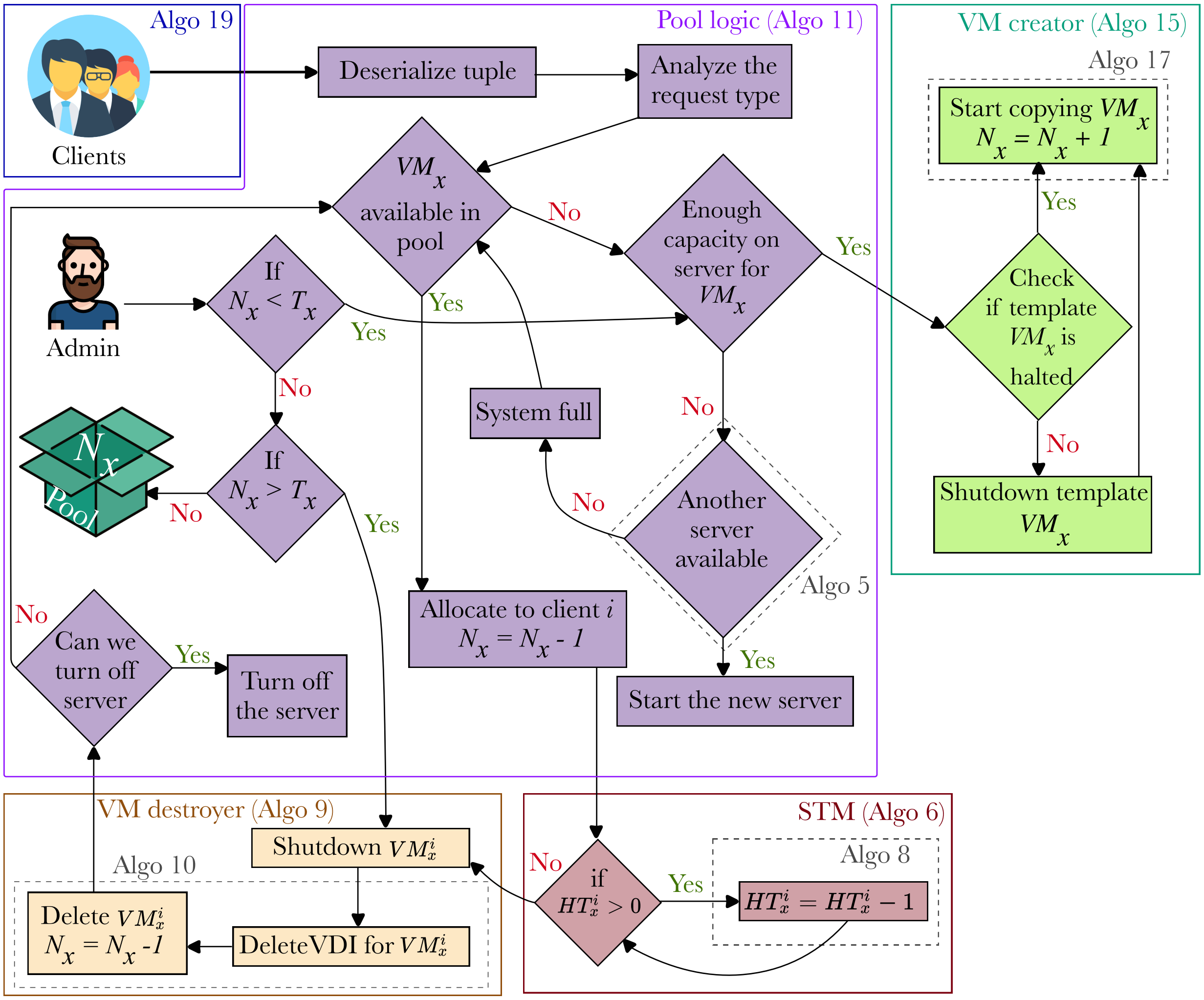


Figure 3.6: Flowchart for the reactive system

and running, the VM destroyer sends a connection request to the XenServer. If the incoming request is accepted by the server, then a socket server starts on port x. Now, the VM creator also sends a connection request to the server and if the request is accepted, then another socket server starts on port y. Similarly, the STM and the pool logic send a connection request to the server and a socket server is turned ON on port z and s respectively. At this time, all the socket server ports are running. Now VM destroyer sends client socket request from port x to port y and establishes a client-server connection. Similarly, a client-server connection is established between port y-z and z-s respectively. Below is the order in which different daemons are started in the pool manager:

$$\text{VM Destroyer} \rightarrow \text{VM creator} \rightarrow \text{Service Time Manager} \rightarrow \text{Pool Logic}$$

Now, since all the connections are successful, the pool manager is ready to handle the incoming client requests.

3.2.2 Pool Logic

The pool logic of the reactive system is different from the allocator logic discussed in Section 3.1.2. It has two different threads running in parallel and has an additional client-server connection to the VM creator. The pool logic is the brain of the pool manager. Once the pool logic has established a successful connection with the Citrix XenServer, then it creates a connection to accept client requests. The pool logic further checks for the connection with the STM and the VM creator. If the connections are well established, it calls two methods named *RunPool* and *ClientAllocator* respectively in parallel as shown on Line 12 of Algorithm 3.11. If the connection was not successfully established, an error message is displayed on the console. Algorithm 3.11 shows the

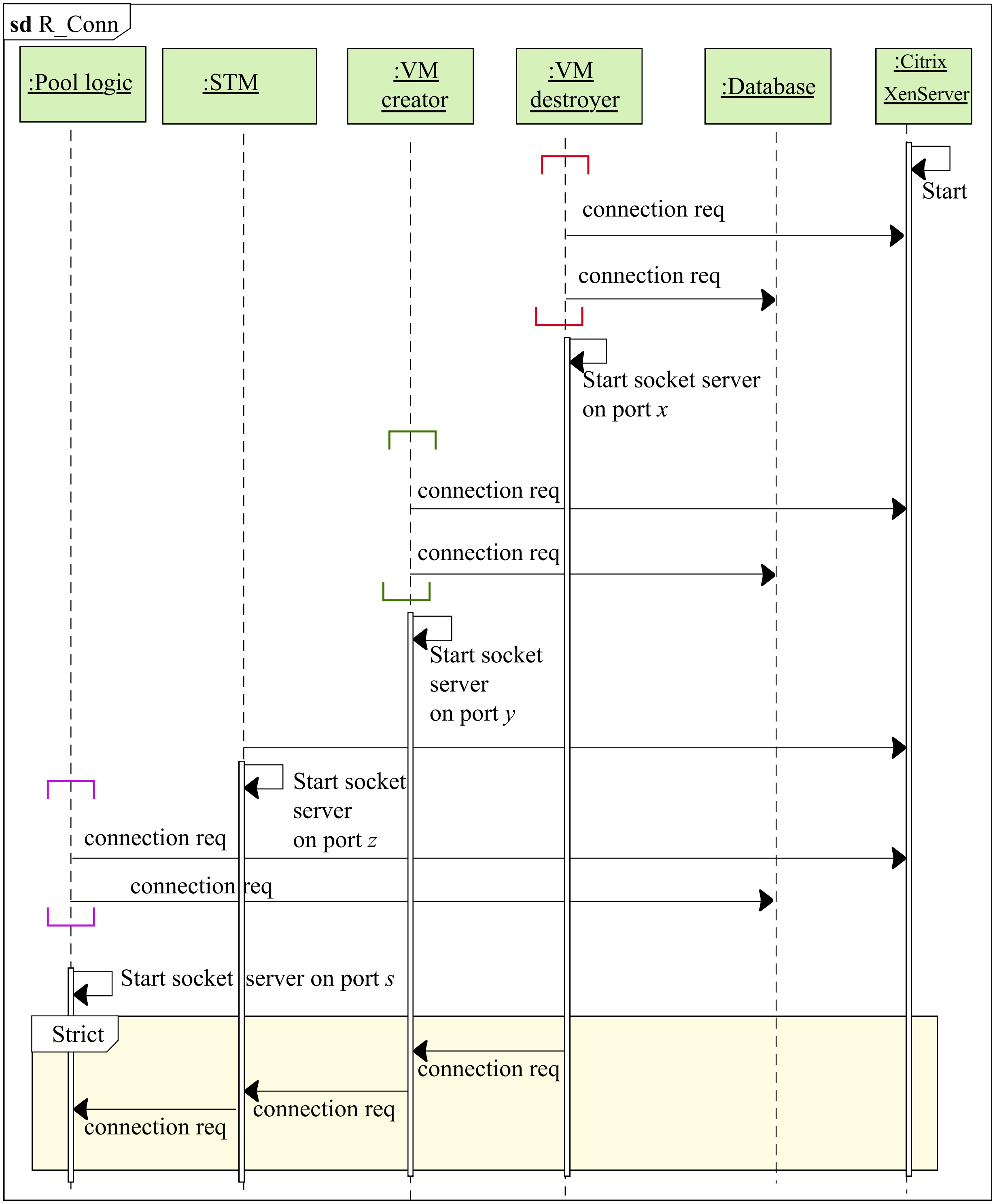


Figure 3.7: Initial connection setup for the reactive system

pseudo-code of the pool logic for the reactive system.

The *RunPool* method makes a call to another method called *CreateAllIntialVMS* explained with the help of Algorithm 3.12. This method takes Session ID and the threshold values of both type of virtual machines (T_A and T_B) as inputs and is responsible to ensure that a fixed number of virtual machines is present in the pool in the halted state. First, the number of standby machines of both Type A (N_A) and Type B (N_B) are statically set to zero. Then a *for* loop runs on Line 3 and gets the list of all the virtual machines present in the server. Next, for all the virtual machines found, it checks for the power state (which should be *halted*)and the *VM type*. Now, another *for* loop at Line 12 compares the values of the threshold (T_x) to the number of virtual machines in the server on standby (N_x). If the T_x value is greater than the value of N_x, then the number of virtual machines that need to be created is found by subtracting N_x from T_x and a request is sent to the VM creator to create the virtual machines. If the value of T_x is less than the value of N_x, then the number of virtual machines that need to be destroyed are found by subtracting T_x from N_x and a request is sent to the VM destroyer to delete the required virtual machines.

Further, another method called *CheckWhenToStartClientHandlingServer* is called whose responsibility is to check whether a minimum number of the virtual machines is available or not. This is required as virtual machines are being created by the VM creator daemon. This function makes sure to start the client request handling server when the minimum number of standby virtual machines have been created and are available in standby mode. The client request handling server receives requests from clients and then appends them to a FIFO queue called *ClientsToBeServed*. Then, for each client at the

Algorithm 3.11: ReactivePoolLogic

```
Input: Login credentials for XenServer & Service time manager & VM destroyer
       needs to be up and running & $T_A$ & $T_B$
Output: VMs are getting allocated and creation requests are sent to VM creator
Try to login into XenServer                                  ▷ using valid credentials
if connection with XenServer ← True then
    S1 ← Create serversocket object to receive client requests
    C1 ← Create clientsocket object to send connection request to STM
    C2 ← Create clientsocket object to send connection request to VM creator
    ClientsToBeServedArray ← null
    $\mathrm{VM_A}$ Name ← ‘VMType-A’
    $\mathrm{VM_B}$ Name ← ‘VMType-B’
    $\mathrm{VM_A}$ Template Name ← ‘Template-A’
    $\mathrm{VM_B}$ Template Name ← ‘Template-B’
    if Pool logic is connected to STM then
        while Pool logic and STM connection is persistent do
            Call $RunPool(Session_{id})$                     ▷ On thread #1
            Call $ClientAllocator(Session_{id})$             ▷ On thread #2
        end
    end
else
    Handle exception and show error message
end
```

head of the *ClientsToBeServed* queue calls the *ClientAllocator* function.

On the second thread in Algorithm 3.11, a method named *ClientAllocator* is called and is explained in Algorithm 3.13. This method is solely responsible for allocating the virtual machines to the incoming client requests. First, the earliest request in the *ClientsToBeServed* queue is picked up and information such as Client ID and VM Type is obtained. This information is further sent to a method called *AllocateAndStartVM* and the request is removed from *ClientsToBeServed* queue.

The method *AllocateAndStartVM* is explained in Algorithm 3.14. The first step in the method is to create a list named *AllocatedVMList*. This list has the information of all the virtual machines that were allocated previously. It is mandatory to ensure that no virtual machine is reallocated. Let us consider an example to explain how can a virtual machine be reallocated after it has finished serving a user an destroy process is in effect. Let's assume virtual machine 'XX' was allocated to client 'C1' at time 'T1'. After some time, 'C1' releases the virtual machine and steps to destroy the virtual machine are initiated. As mentioned in Section 3.1.4, the VM destroyer involves a set of steps, where the first step is to shut down a running VM in order to destroy the VDI and VM. Now, let's assume that client 'YY' comes in requesting a VM at the same time when VM 'XX' was in the process of deletion (meaning it was in halted state). This creates a confusion for the pool logic as the condition that the VM should be in halted state (meaning the virtual machine in available in the pool) is satisfied and pool logic reallocates the particular virtual machine. This results in an error due to the fact that same VM will be getting allocated and deleted at the same of time. Thus, to avoid confusion, the list *AllocatedVMList* comes in handy. Now pool logic finds, all the virtual machines that

Algorithm 3.12: CreateAllInitialVMS

```
Input: $Session_{id}$ & $T_A$ & $T_B$
Output: Minimum number of VM's have been created
Set $N_A$ , $N_B \leftarrow 0$                    ▷ where, $N_A$ represents # standby VM of Type-A
vms ← session.xenapi.VM.getAllRecords()
foreach vmRef ∈ vms do
    vmRec ← vms[vmRef]
    if vmRec[‘isAtemplate′] ← False & vmRec[‘isControlDomain′] ← False &
      vmRec[ ‘powerState′] ← ‘Halted′ & vmRec[‘nameLabel′] ← ‘VMType-A′
      then
        $N_A \leftarrow N_A + 1$
    end
    if vmRec[′isAtemplate′] ← False & vmRec[‘isControlDomain′] ← False &
      vmRec[ ‘powerState′] ← ‘Halted′ & vmRec[′nameLabel′] ← ‘VMType-B′ then
        $N_B \leftarrow N_B + 1$
    end
end
foreach $T_x \leftarrow$ A & B do
    if $T_x > N_x$ then
        $C_x = T_x - N_x$                 ▷ where, $C_x$ is # VM's need to be created of type-x
        for x $\leq C_x$ do
            $N_{vm} \leftarrow 1$                         ▷ $N_{vm}$ = 1 means , create 1 VM
            Tuple ← [$VM_{Type}$ ,$N_{vm}$]
            Serialize tuple and send to VM creator
            x ← x + 1
        end
    else
        $D_x = N_x - T_x$                 ▷ where, $D_x$ is # VM's need to be deleted of type-x
        for y $\leq D_x$ do
            $N_{vm} \leftarrow 1$                          ▷ $N_{vm}$ = 1 means, create 1 VM
            DestroySingleVM($Sessionid$, $VM_{Type}$)
            y ← y + 1
        end
    end
end
```

are present in the server and are not present in *AllocatedVMList*. If a virtual machine is found in the server, then it is started and allocated to the client. The start time of the virtual machine is saved and serialized with other information and passed to STM on one thread. On the second thread, a tuple with VM_x is passed to VM creator to create a virtual machine of a similar type as allocated. This step ensures that the N_x is always equal to T_x.

Algorithm 3.13: ClientAllocator

```
Input: $Session_{id}$ & Client Request
Output: Minimum number of VM have been created
cursor, db ← connection to allocation database    ▷ From the same thread
while true do
    if ClientsToBeServed Queue $\neq$ Empty then
        Get earliest request tuple
        Get $ClientID^i$, $H^i_T$ and $VM_x$
        IsRequestSent ← False
        AllocateAndStartVM($Session_{id}$, $ClientID^i$, $VM_x$, $H^i_T$, $cursor$, $db$,
          $IsRequestSent$)
        Remove request from Queue
    end
end
```

3.2.3 Reactive VM Creator

So far we have discussed how virtual machines are allocated to incoming clients. Now, we need to describe how virtual machines are created. Figure 3.8 explains the concept

Algorithm 3.14: AllocateAndStartVM

```
Input: $Session_{id}$, $ClientID^i$, $VM_x$, $H^i_T$, $cursor$, $db$, $IsRequestSent$
Output: VM has been allocated or waiting to be allocated
AllocatedVMList ← null                                  ▷ For avoiding the ambiguous condition
cursor.execute('select uuid from allocationTable')
rows ← cursor.fetchall()
foreach row in rows do
    AllocatedVMList ← row[0]
end
vm, vms ← None, session.xenapi.VM.getAllRecords()
foreach vmRef ∈ vms do
    if vmRec['uuid'] not in '$AllocatedVMList$' & vmRec['isAtemplate'] ← False &
      vmRec['powerState'] ← 'Halted' & vmRec['nameLabel'] ← '$VM_x$' then
        vm ← vmRef
        session.xenapi.VM.start(vm, False, True)
        uuid ← vmRec['uuid']                                 ▷ saving for reference
        $VM_{StartTime}$ ← now()
        Form time tuple to send it to Service time manager
        Serialize and send tuple via $C_1$ object                      ▷ step # 1
        Tuple ← [$VM_{type}$, 1]
        Serialize and send request to VM creator via $C_2$ object      ▷ step # 2
        Write $VM_{StartTime}$, $UUID$ and $ClientID^i$ to allocation database   ▷ step # 3
        break
    end
end
if $vm$ ← None & $IsRequestSent$ ← False then
    Tuple ← [$VM_{type}$ , 1]
    Serialize and send tuple via $C_2$ object
    $IsRequestSent$ ← True
end
if $vm$ ← None & $IsRequestSent$ ← True then
    AllocateAndStartVM($Session_{id}$, $ClientID^i$, $VM_x$, $H^i_T$, $cursor$, $db$, $IsRequestSent$)
      ▷ recursive calling
end
```

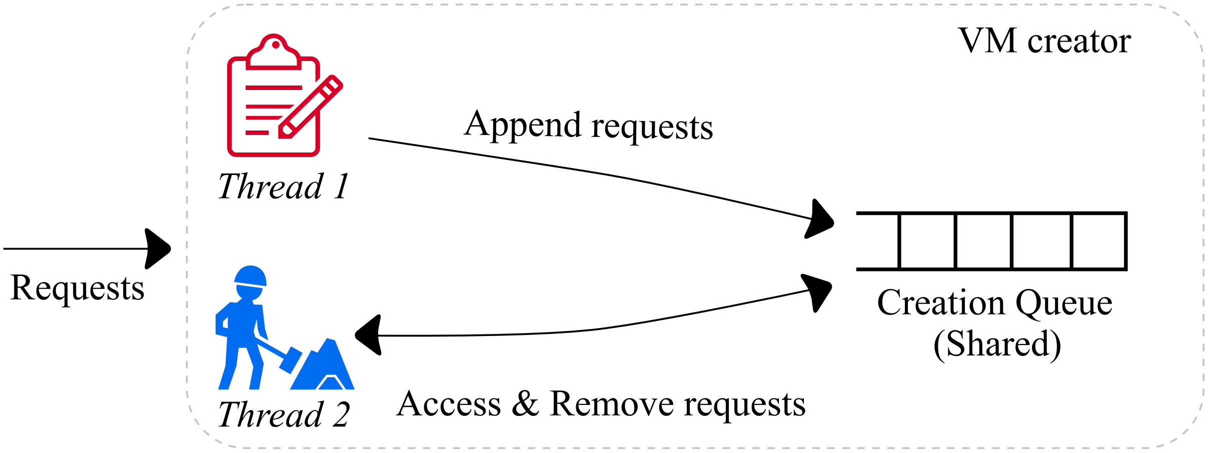


Figure 3.8: Architecture showing working of VM creator

of VM creator of the reactive system. There, are two threads working in parallel and one shared queue. On Thread 1, a method named *CreationRequestHandler* is called and on Thread 2, method *VMCreatorWorker* is called. *CreationRequestHandler* receives the requests from the pool logic and deserializes them and appends them to the *Creation Queue* in FIFO order. Further, the VM creator daemon obtains the information about Session ID, Type of VM and sends it to another method called *VMCreatorWorker*.

```
Algorithm 3.15: ReactiveVMCreator
  Input: Login credentials for XenServer, $VMTemplate_x$, $MemoryThreshold_x$,
         $StorageThreshold_x$
  Output: Stops the requests after service is over and writes to database
  Create server socket object and wait of connection from pool manager
  $SessionId$ ← Try to login to XenServer
  if Login to XenServer ← successful then
      Creation Queue ← null
      Start CreationRequestHandler()                        ▷ on Thread 1
      Start VMCreatorWorker($Session_{id}$)                 ▷ on Thread 2
  end
```

Algorithm 3.16: CreationRequestHandler

```
Input: Server socket reference
Output: Receives the requests and handles them appropriately
while true do
    Running ← true
    ClientSocket, address ← serversocket.accept()
    while running do
        Request ← socket.receive()
        if request != null then
            De-serialize the request
            Append request to shared queue
        end
    end
end
```

The *VMCreatorWork* method (see Algorithm 3.17), picks the first request from the queue and then makes a call to function *CreateSingleVM* which starts copying a new virtual machine. Once the process of copying a virtual machine is completed, the variable *isVmCreated* is set to true and the database is updated. The function *CreateSingleVM* is explained in detail in Section 3.1.2 (Algorithm 3.3).

The service time manager is responsible to keep track of all active clients and is used to decrease the simulated time by one second. The STM for the reactive system works the same way as in the baseline system and is explained in Section 3.1.3.

The VM destroyer also has the same responsibility i.e. to delete the virtual machine and the disk image and the steps to do so are explained in Section 3.1.4.

3.3 Proactive System

Until now we have discussed how the virtual machines are allocated in the baseline system and the reactive system. We saw that the reactive system solves issues in the baseline system by maintaining a pool of virtual machines and allocating them to clients upon request which significantly decreases the client waiting time. This point will further be discussed in the next chapter. The proactive system is the novel idea and prominent system. It is a smart and adaptive system which learns the client arrival behavior over time.

Algorithm 3.17: VMCreatorWorker

```
Input: $SessionID$
Output: Creates a VM and updates it to database successfully or VM not created
        due to various reasons
dbRef, cursorRef ← connect to vm_creation_database while true do
    if creation queue = null then
        vm ← Get the first element from queue
        $isVmCreated, UUID, VM_x, Creation_{StartTime}, Creation_{EndTime}$ ←
          CreateSingleVM($SessionID$, $VMToBeCreated$)
        if isVmCreated ← true then
            Update database($UUID, VM_x, Creation_{StartTime}, Creation_{EndTime}$ )
            Dequeue the request from queue
        else
            Dequeue the request from queue
        end
    end
end
```

Figure 3.9 shows an overall architecture of the proactive system. All the daemons: Client, Pool logic, STM, VM creator and VM destroyer works the same way as explained for the reactive system in Section 3.2. The only difference between the reactive and proactive system is the new daemon called Predictor. Please note that in a real implementation all the modules except the client and the STM modules will need to be incorporated. The client and the STM modules were required only to evaluate the system.

1. Predictor: This module is the heart of the proactive system. It is responsible to read and analyze the historical client arrival data and makes a prediction for the required number of virtual machines in a specific period of time. To make a prediction, it uses an autoregressive prediction model. Autoregressive models are a fundamental class of time series model. They take the advantage of the fact that time series data is sequential and order is very important to maintain. We are using this model because our client arrival data is in time series format and autoregressive is remarkably flexible at handling a wide range of different time series patterns. In this model, we forecast the number of required virtual machines by using a linear combination of the past values. The term autoregression indicates that it is a regression of the variable against itself. Below is the equation that gives an idea of an autoregressive model:

   $$y_t = c + \phi_1 y_{t-1} + \phi_2 y_{t-2} + \cdots + \phi_p y_{t-p} + e_t \tag{3.3}$$

   where, c is a constant, e_t is the white noise and y_t are the lagged values.

   The coefficients ϕ_1, ϕ_2, …, ϕ_p defines the importance of the lagged values. The

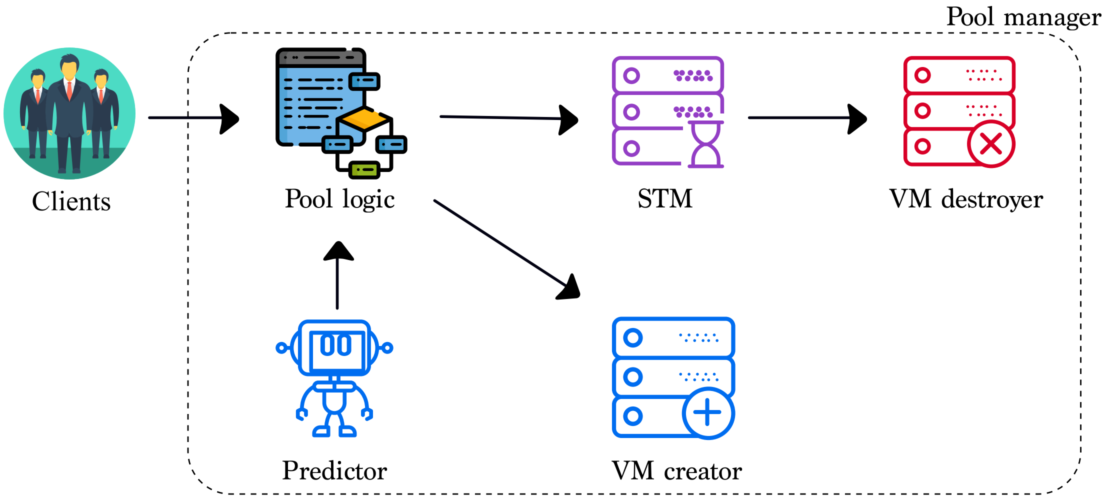


Figure 3.9: Architecture showing different components of proactive system

largest coefficients are multiplied by the observations in the time series. The Equation (3.3) states that our prediction y_t is heavily dependent on the most recent values and least on the most lagged value. However, the coefficients don't necessarily have to follow this pattern. The main point to note here is how a time series is correlated to its past values.

Figure 3.11 shows the flowchart for a clients request in the baseline system and how are different algorithms linked to each other.

3.3.1 Initial Connection Setup of the Proactive System

Figure 3.10 illustrates the initial connection setup of the pool manager of the proactive system. It is very important to ensure that all connections among the different daemons are well established. The first step is to turn On the Citrix XenServer so it can accept all the incoming request from the pool manager. Next, we start the predictor module so that it can start reading the historical client arrival data. The system proposed is based on the client-server socket. Once both the Citrix XenServer and the predictor module are up and running, VM destroyer sends a connection request to the Citrix XenServer. If the request is accepted by the Citrix XenServer, it sends an acknowledgment and starts a socket server on port x. Further, the VM creator sends another request to the Citrix XenServer and if it is accepted then another socket server starts on port y. Similarly, a socket server connection is established between the STM and the server on port z and port *p2* respectively. Now that all the socket servers are up and running, so VM destroyer sends a client socket request from port x to port y and a client-server connection is established between VM destroyer and VM creator. In the same way, a client-server connection is established between VM creator - STM and STM - Pool logic. Once all these connection are done another client-server connection is made between the predictor and pool logic on port *p1*. When all the connections are successful and all the modules are up and running, the pool manager starts accepting the incoming clients requests. Below is the order in which different modules are started:

$$\text{VM destroyer} \rightarrow \text{VM creator} \rightarrow \text{STM} \rightarrow \text{Pool logic} \leftarrow \text{Predictor}$$

3.3.2 Pool Logic

The pool logic of the proactive system is slightly different from the reactive system we studied in Section 3.2.2. It has three different threads instead of two. The first thread receives the incoming request and appends it to a FIFO queue. The second thread picks the earliest request from that queue and sends it to the *ClientAllocator*. On the third thread, the pool logic is connected to the predictor module. The predictor module is responsible for making a forecast of the different threshold values (i.e. T_A and T_B) for a

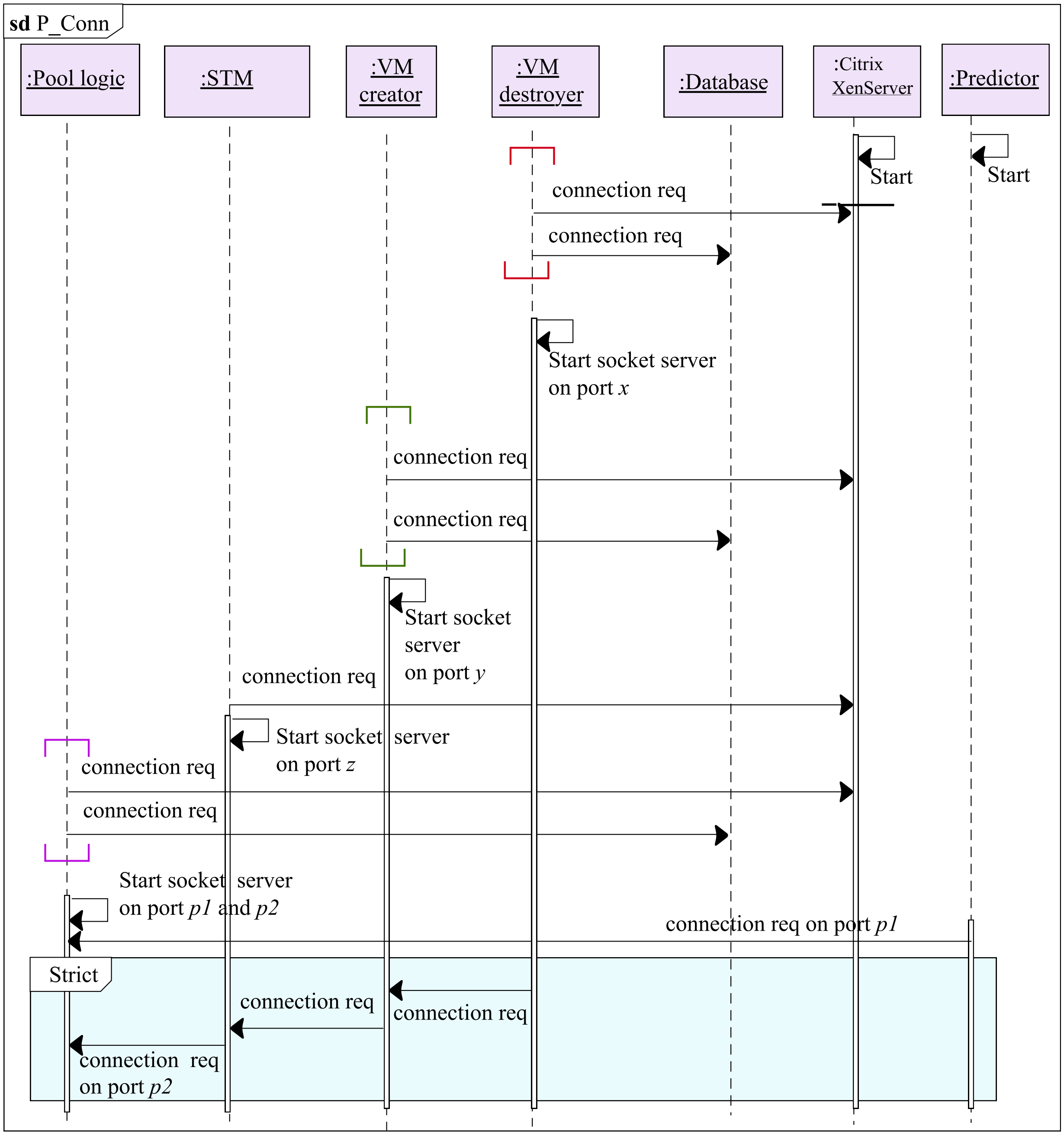


Figure 3.10: Connections between different modules of proactive system

specific time frame. Once, the predictions have been made, then these values are passed to the pool logic where they are used in the method called *CreateAllIntialVMS* explained in Algorithm 3.12.

Algorithm 3.18 explains the concept of how the predictor module works. This module takes the input as the past values of clients arrival data and the type of service they used. A connection to the database is made from where it reads the historical data. Then, it ensures that there is a connection with the pool logic. Now, we create an empty array for Time, Type A virtual machines and Type B virtual machines respectively. At Line 8, we calculate the interval length by diving the time to go back by the interval gap. For example, let's assume we want to make a prediction for the next 15 minutes at 7:00 pm using the last 1 hour of data. Using this example, the time to go back becomes 60 minutes and with a time interval gap of 5 minutes, we obtain an interval length of 12. Now, this means 12 different intervals of time between 6:00 pm to 7:00 pm i.e. 6:00-6:05, 6:05-6:10, . . . , 6:55-7:00. These intervals of time are now appended to the *TimeArray* using a *for* loop at Line 11. At the end of the *for* loop, we reverse the order of the time array. The array formed in the previous step has both types of virtual machines. Next, we form separate arrays for both Type A and Type B virtual machines by appending them to arrays named *ClientAarray* and *ClientBarray* respectively as shown on Line 15. The reason to form separate arrays for each type of clients is that the autoregression model can be applied to symmetrical data. At Line 22, we use an in-build stats model that uses the *Fit* function to train on the data in the lists *ClientAarray* and *ClientBarray*. The *maxlag* is calculated by subtracting the predicted length from the interval length. From the above example, we found the interval length to be 12, now to find the predicted length we divide 15 minutes by 5 minutes. Therefore, the predicted length becomes 3 and maxlag becomes 9. Once the training has been done on the data, another method is used on Line 23 to make a prediction based on the trained data.

Further, these predicted values are sent to the pool logic using the client socket port where they replace the previous values. Predicting the threshold values make the pool size dynamic. So, to maintain the pool size with respect to the threshold values of the specific time frame, a function named *ThresholdManager* is called. The job of this function is to compare the new threshold value to the previous ones. If the new values are greater than the previous values, then a call to method VM creator is made, which starts copying the desired number of virtual machines and adds them to the pool. If the new threshold values are less than the previous values, then a call to the method VM destroyer is made, which starts to delete the desired number of virtual machines from the pool. The destroyer function will only delete virtual machines that are in halted state and not allocated to clients.

Figure 3.11 shows the flowchart for a clients request in the baseline system and how are different algorithms linked to each other.

Algorithm 3.18: Predictor

```
Input: $Time_{ToGoBack}$ & $Time_{toPredict}$ & $Time_{IntervalGap}$ & $Time_{sleep}$
Output: Minimum number of VMs have been created
$\mathrm{Predicted_{TA}} \leftarrow 0$
$\mathrm{Predicted_{TB}} \leftarrow 0$
Cursor ← database connection to Allocation Database
$\mathrm{CS_1}$ ← Client socket object for sending predicted values to pool manager
TimeArray ← null
ClientAarray ← null
ClientBarray ← null
IntervalLength ← $Time_{ToGoBack}$ / $Time_{IntervalGap}$
Now ← datetime.datetime.now()
PastTime ← now + datetime.timedelta(minutes = - $Time_{ToGoBack}$)
for x ∈ range(1,IntervalLength ) do
    TimeArray.append(Now + datetime.timedelta(minutes =- $Time_{IntervalGap}$ * x))
end
Reverse the TimeArray
for i ∈ range(1,IntervalLength ) do
    $\mathrm{A_{Count}}$ ← select count(*) from tableName where vmType = ‘TypeA’ and
      arrivalTime BETWEEN TimeArray[i-2] AND TimeArray[i-1]
    $\mathrm{B_{Count}}$ ← select count(*) from tableName where vmType = ‘TypeB’ and
      arrivalTime BETWEEN TimeArray[i-2] AND TimeArray[i-1]
    Append $\mathrm{A_{Count}}$ to ClientAarray
    Append $\mathrm{B_{Count}}$ to ClientBarray
end
$\mathrm{AR_A}$ ← AR(ClientAarray)
$\mathrm{P_A}$ ← $\mathrm{AR_A}$.fit(maxlag, ic=’hqic’, trend=’nc’)
$\mathrm{Predicted_{TA}}$ ← $\mathrm{P_A}$.predict(start , end , dynamic=True )
$\mathrm{AR_B}$ ← AR(ClientBarray)
$\mathrm{P_B}$ ← $\mathrm{AR_B}$.fit(maxlag, ic=’hqic’, trend=’nc’)
$\mathrm{Predicted_{TB}}$ ← $\mathrm{P_B}$.predict(start , end , dynamic=True )
send $\mathrm{Predicted_{TA}}$ & $\mathrm{Predicted_{TB}}$ to Pool Logic using $\mathrm{CS_1}$
Thread.sleep($Time_{sleep}$)
```

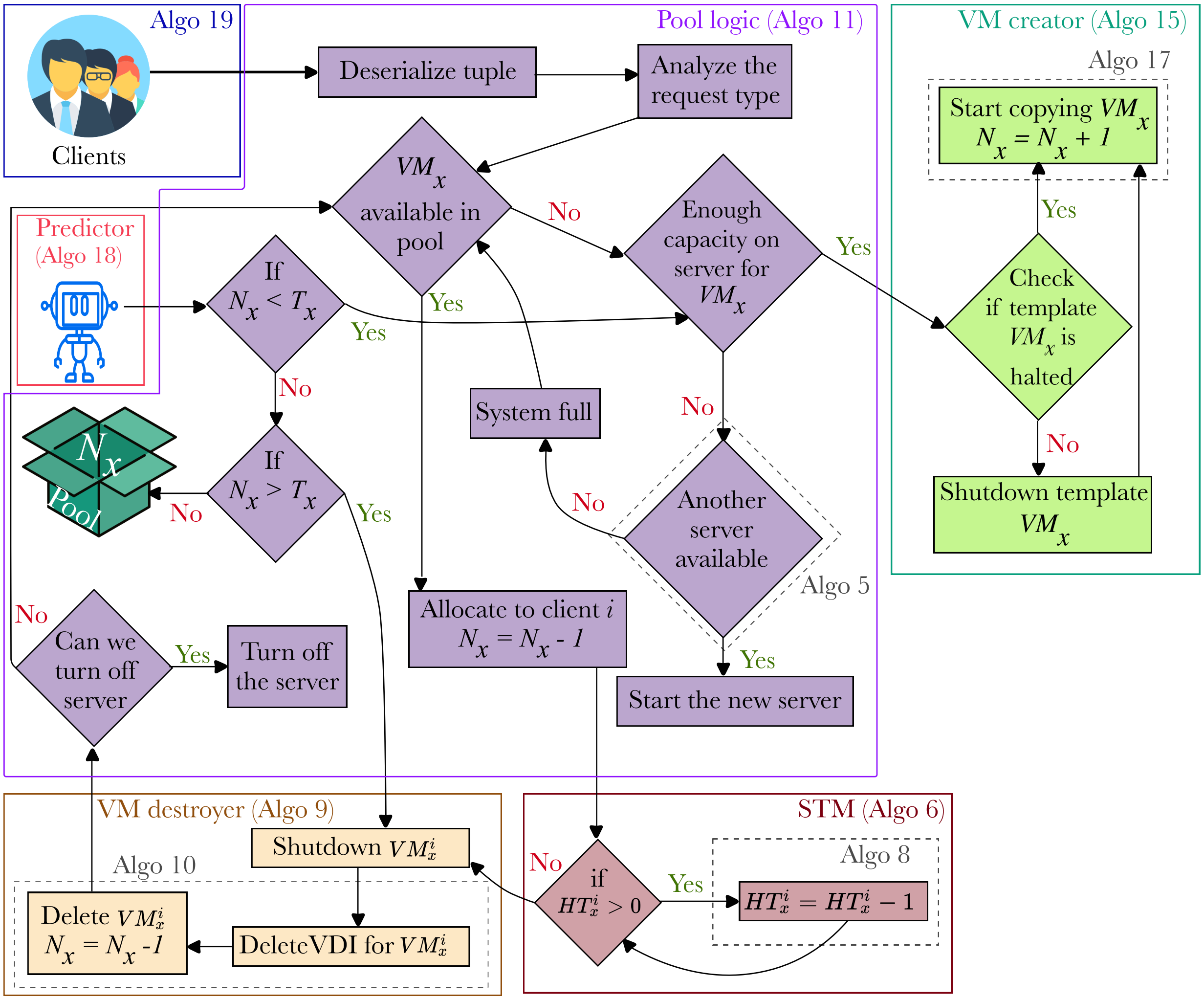


Figure 3.11: Flowchart for the proactive system

Chapter 4

Performance Analysis

This chapter describes the use cases and experiments that are conducted to study the proposed systems. The request generator is described first. Next, the workload and systems parameters are outlined. Finally, the results of the experiments and the system behavior are evaluated and summarized.

4.1 Request Generator

The request generator is a module designed to simulate the behavior of the client arrival in the systems. It generates synthetic data which is then passed to the allocator in the baseline system and to the pool manager in the reactive and proactive systems. The data tuple that corresponds to each generated request comprises of the following components and is elaborated in Table 4.1.

$$< ClientID^i, H_T^i, A_T^i, VM_x^i >$$

Each request corresponds to a specific client (i^{th} client) requesting a VM. The generation of the client requests is described next with the help of the pseudo-code given in Algorithm 4.1. The first step is to successfully make a connection to the database (Line 1). Upon a successful connection, we need to provide a few inputs parameters such as:

Table 4.1: Data tuple sent by the request generator

Symbol	**Description**
$ClientID^i$	Unique client ID of i^{th} client
H_T^i	Virtual machine hold time of i^{th} client
A_T^i	Arrival time of i^{th} client
VM_x^i	Type of VM requested by i^{th} client

- the probability to recieve a request for a specific VM type. In our case, since we consider 2 different VM types, p_A and p_B are provided (see lines 2-3).
- the arrival rate at which requests are being generated (Line 5).
- the service rate (Line 5) so that we can simulate how long the VM will be used for (referred to as VM hold time).
- the simulation time (Line 6) for which the experiment will be run.

From, the value of the arrival rate, the inter-arrival time between the consecutive requests is calculated using the inbuilt function of python called *expovariate* (Line 15). Similarly, the *expovariate* function is also used to calculate the virtual machine hold time for each request (Line 16). Next, the algorithm decides the type of virtual machine by using the *choice* function and based on the output, the VM type is set to VM_A or VM_B (Line 18). Now, a tuple is created with all the values obtained, serialized and sent to the allocator (in the baseline system) or the pool manager (in reactive and proactive systems) respectively. These steps (line 14-29)are repeated each time a new client request is generated until the simulation end time.

Algorithm 4.1: RequestGenerator

```
if database connection ← successful then
    $p_A$ ← $x$ where x ∈ [0,1]
    $p_B$ ← 1-$x$
    $\lambda$ ← arrival rate
    $\mu$ ← service rate
    ExpectedEndTime ← simulation time in minutes
    Now ← datetime.datetime.now()
    Difference ← datetime.timedelta(minutes = $Time_{end}$)
    $T_{end}$ ← now + difference
    ClientSocket ← socket object
    ClientSocket.connect('ServerIP','Client Port number')
    ClientID ← 1
    if clientSocket connection ← successful then
        while datetime.datetime.now() ≤ $T_{end}$ do
            InterArrivalTime ← random.expovariate($\lambda$)
            HoldTime ← random.expovariate($\mu$)
            RequestTime ← datetime.datetime.now()
            var ← numpy.random.choice(numpy.arrange(1,3), $P_A$, $P_B$)
            if $var$ ← 1 then
                VMType ← A
            else
                VMType ← B
            end
            ClientTuple ← [ClientID, RequestTime, InterArrivalTime, ServiceTime]
            Send request to allocator (Baseline system) OR
            Send request to pool manager (Reactive & Proactive systems)
            Save this data to arrival database
            ClientID ← ClientID + 1
            Time.sleep(InterArrivalTime)
        end
    end
end
```

4.1.1 Workload and System Parameters

During experiments, each workload and system parameter is varied in a given experiment while the others were held at their default values. The various values for both workload and system parameters used in the experiments are shown in Table 4.2. The values in bold correspond to the default value for the parameter. The following are the workload parameters used while evaluating the performance of the systems:

- Arrival Rate (λ): It is the rate at which clients arrive in the system. We are using the Poisson arrival process, which implies that the inter-arrival times are exponentially distributed.
- Virtual Machine Hold Time (H_T): It is the amount of time a client uses the services of the virtual machine. This time is also exponentially distributed.
- Probability of VM Type (p_x): This parameter is used to select the type of virtual machine. It is needed since there are two types of services a client can request (type A and type B).

The following system parameter is used while evaluating the system performance:

- Pool Size (P_S): This parameter is only used in the reactive system because we are maintaining a pool of virtual machines with a static threshold value for both types of virtual machines.

Table 4.2: Workload and system parameters

Parameter	**Values**	**Units**	**Distribution/Process**
λ	30, 40, 50, **60**, 68, 100, 110	Clients/Hour	Poisson Process
H_T	5, **10**, 15, 20, 25, 30	Seconds	Exponential Distribution
p_x	0, 0.25, **0.5**, 0.75, 1	-	Constant
P_S	10, 20, **30**, 40, 50, 56	-	Constant

4.2 Performance Metrics

Two performance metrics are used to compare the performance of the three systems: Mean waiting time ($\overline{\mathrm{W}}_T$) and mean idle time ($\bar{\mathrm{I}}_T$). The computation of these metrics is based on other auxiliary performance metrics and time stamps generated during the experiments as discussed below. Figure 4.1 represents the different time stamps for the baseline system and Figure 4.2 shows the different time stamps for both the reactive and proactive systems.

- Total clients (C_{Tot}): It is the total number of clients that arrived in the system.
- Queuing time (Q_T): It is the average time for which a client waits for the services in a FIFO queue.
- Creation start time (C_{ST}): Time at which the creation of a virtual machine starts.
- Creation end time (C_{ET}): Time at which the creation of a virtual machine is completed.

- Creation time (C_T): It is the time taken to create a virtual machine in Citrix XenServer. It is obtained as:

$$C_T = C_{ET} - C_{ST} \tag{4.1}$$

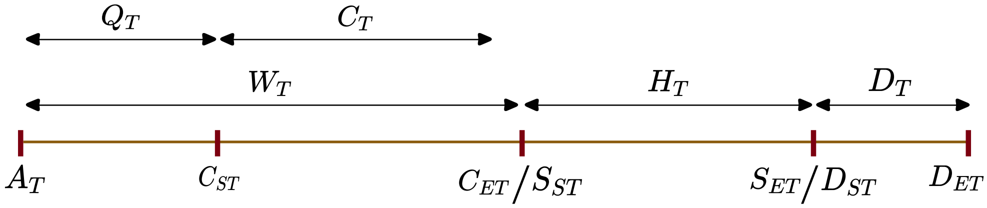


Figure 4.1: Time stamps for the baseline system

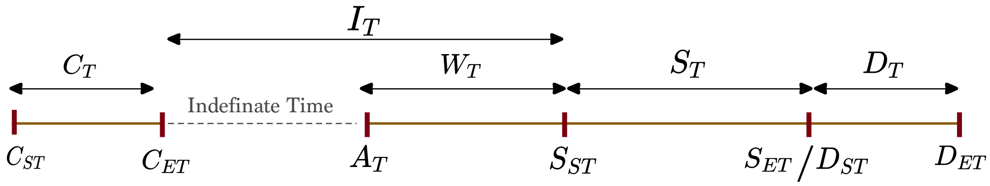


Figure 4.2: Time stamps for the reactive and proactive systems

- Service start time (S_{ST}): It is the time at which the virtual machine is allocated to the client who starts working on the virtual machine.

- Service end time (S_{ET}): It is the time at which a client has finished working on the virtual machine and terminates the session.

- Destruction start time (D_{ST}): The time at which the process of deleting a virtual machine starts.

- Destruction end time (D_{ET}): The time at which the process of deleting a virtual machine is completed.

- Destruction Time (D_T): It is the time taken to destroy the virtual machine and the disk image associated with it. It is obtained as:

$$D_T = D_{ET} - D_{ST} \tag{4.2}$$

- Waiting Time (W_T): It is the time a client has to wait before a virtual machine is allocated to it. Waiting time in the baseline system is calculated as the sum of queuing time and creation time.

$$W_T = Q_T + C_T \tag{4.3}$$

Waiting time in both reactive and proactive systems is only equal to the queuing time since the systems have a pool of pre-created VMs.

$$W_T = Q_T \tag{4.4}$$

- Idle time (I_T): Idle time of a virtual machine is defined as the time difference between the service start time and creation end time. In other words, it is the time for which the virtual machine was idle in the pool (not being used by the client) in a halted state. Please note, there is no idle time in the baseline system beacuse virtual machine is only created when the client requests a virtual machine.

$$I_T = C_{ET} - S_{ST} \tag{4.5}$$

- Mean waiting time ($\overline{W}_T$): The average time a request spends in the queue before the virtual machine is allocated.

$$\overline{W}_T = \frac{\sum_{i=1}^{N} W_T}{C_{Tot}} \tag{4.6}$$

- Mean idle time ($\bar{\mathrm{I}}_{\mathrm{T}}$): It is the mean time for which a virtual machine is idle in the pool.

$$\bar{\mathrm{I}}_T = \frac{\sum_{i=1}^{N} \mathrm{I}_T}{\mathrm{C}_{Tot}} \tag{4.7}$$

4.2.1 Experimental Setup

All the three systems are built on the PyCharm Integrated Development Environment (IDE) using the python programming language version 2.7. Proof-of-concept prototypes for the three systems have been tested on a Citrix XenServer deployed on a system comprising a 3.5 GHz CPU with 4 cores and 16 GB of memory using simulated (synthetic) client workload. A module called "Client" is deployed on a system equipped with an Intel Core i7 CPU and 16 GB RAM running the Ubuntu 14.04 operating system. This module is used to generate client requests at the desired rates. A JSON tuple which consists of synthetic workload data such as client id, service type and service time is passed to the pool manager for each request generated.

The performance metrics such as the average waiting time and the average idle time are computed during each experiment. A set of experiments was performed with a duration of 180 minutes for each experiment. Note that running the experiment for 180 minutes ensured that the system was running in a steady state.

For the prediction module, the algorithm makes prediction every 10 minutes using the past data. We found out that making predictions every 10 minutes provides the best trade-off between computation time and the percentage of error.

Accuracy of the Average Waiting Time

The accuracy of the average waiting time determined during the experiments is discussed in Appendix A.

Accuracy of the Prediction Algorithm

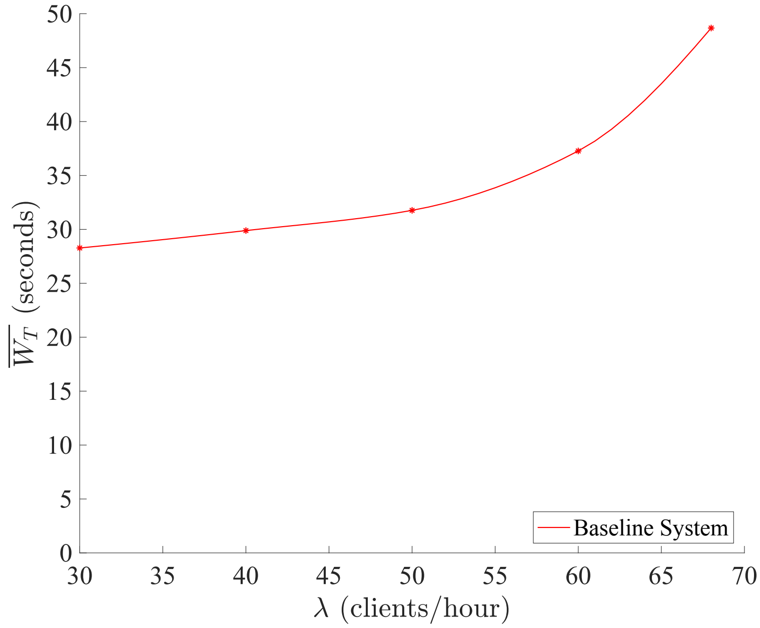
A separate set of experiments was performed to show the accuracy of the prediction as we increase the value of the arrival rate. A discussion of these experiments is presented in Appendix B.

4.3 Experiments for the Baseline System

This section discusses the effect of the different workload parameters described in Section 4.1.1. All the simulations were performed for a duration of 180 and the results (average values)obtained from multiple runs are presented.

4.3.1 Effect of Arrival Rate on the Mean Waiting Time

Figure 4.3 shows the relationship between the average waiting time of the clients in the baseline system and the arrival rates. In this case, the wait time for a client is the sum of the time taken to create a virtual machine (C_T) and the time taken to start a virtual machine (S_{ST}). The figure shows that at λ = 30 clients/hour, the average time required to create a VM and allocate it to clients is 28.276 seconds. But as λ increases, the average waiting time also increases. This is due to the fact that with a higher number of clients arriving in the system, contention for the resources increases. This leads to an increase in the length of the queue which increases the time to serve the clients. To note, if we increase λ to 60 clients/hour, which is twice the value when we started the system, the waiting time increases by more than 35%. On further increasing λ to 68 clients/hour, the system reaches its saturation point as the traffic intensity becomes almost equal to 1 as the number of clients arriving in the system is almost equal to the number of clients being served by the system thus from this point, the average waiting time increases sharply.

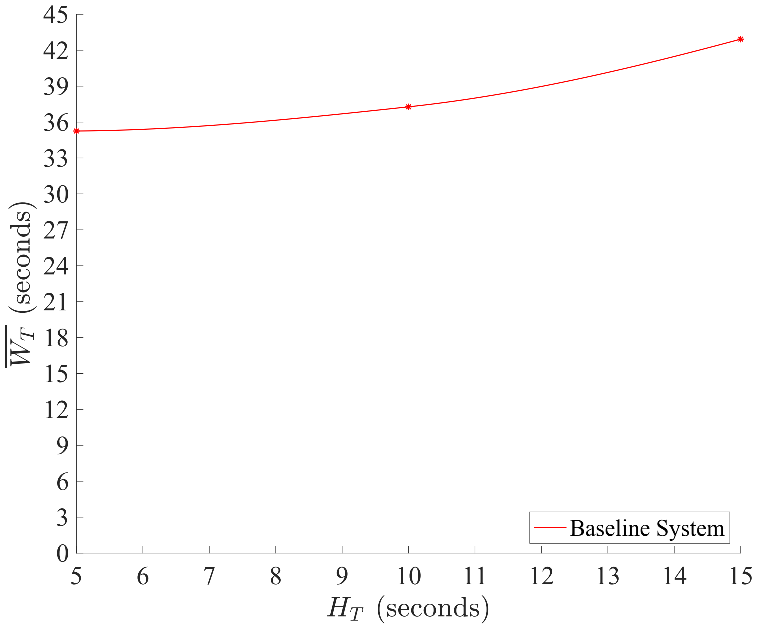


λ	Value
30	28.276
40	29.889
50	31.766
60	37.274
68	48.673

Figure 4.3: Effect of λ on $\overline{W_T}$

4.3.2 Effect of Hold Time on the Mean Waiting Time

Figure 4.4 shows how the virtual machine hold time impacts the average waiting time for the clients. For this experiment, the number of incoming arrivals is fixed at a rate of 60 clients/hour. The virtual machine hold time is varied from 5 to 15 seconds. As we increase the time for which a client uses a virtual machine on the server, the load on the server increases which results in increasing the start time (it is the time between the creation end time and the service start time) of a virtual machine. So, when clients hold the virtual machine for a longer duration, the start time for acquitting a VM increases which result in the increment of the overall average waiting time for the clients.

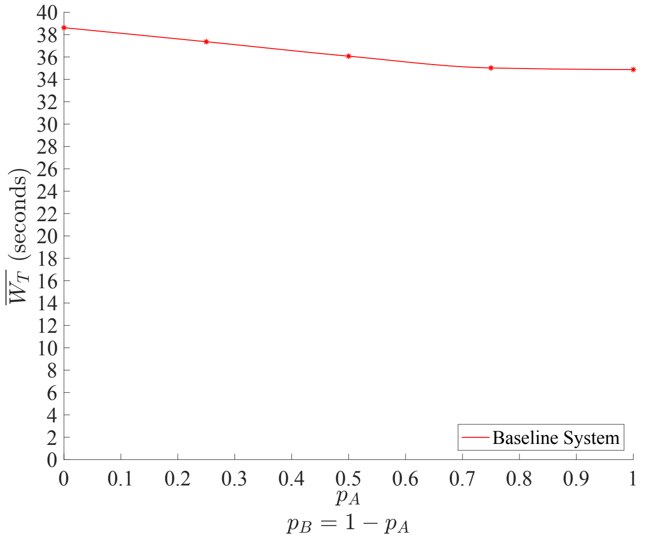


H_T	Value
5	35.253
10	37.274
15	42.926

Figure 4.4: Effect of H_T on $\overline{W}_T$

4.3.3 Effect of Probability of Selecting the type of VM on the Mean Waiting Time

There could be a possibility that the clients coming into the system do not have the same probability of occurrence. Figure 4.5 explains the relationship between the probability of selecting the type of a virtual machine and the average waiting time for clients. When the probability of selecting a virtual machine of type A $p_A = 0$, it means that all the clients coming into the system are demanding a virtual machine of type B. Since the size of the virtual machine of type B is bigger in comparision to type A, the overall waiting time for clients is higher. At p_A = 1, all the clients coming into the system are demanding the virtual machines of type A. Since the size of the virtual machine of the type A is smaller in comparison to the type B, the creation time and the start time is less resulting in a lower overall average waiting time of the clients.



p_A	Value
0	38.619
0.25	37.364
0.5	36.074
0.75	35.021
1	34.877

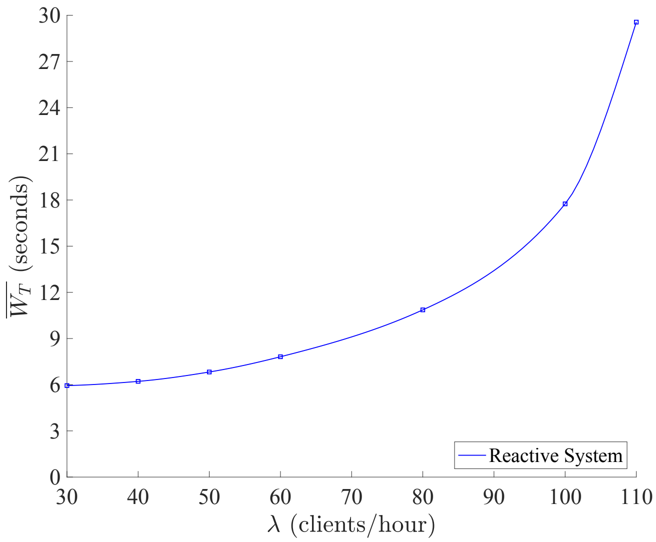
Figure 4.5: Effect of p_x on $\overline{W}_T$

4.4 Experiments for the Reactive System

This section explains the effect of different workload parameters on the average waiting time of the clients and the average idle time of virtual machines in the reactive system. As a reminder, the reactive system has a fixed size of the pool of VMs that are pre-created and ready to be assigned to clients. The default pool size for the reactive system is set at 30 where the number of virtual machines of each type is 15 (VM_A = VM_B = 15). The simulations were performed for a period of 180 minutes.

4.4.1 Effect of Arrival Rate on the Mean Waiting Time

The average waiting time of clients is observed to increase with an increase in the arrival rate for the clients in the reactive system (see Figure 4.6). In the case of the reactive system, the waiting time for the client is only the time it spends in the queue. As mentioned in Section 3.2, a fixed number of virtual machines are created a-priori and are placed in a pool on a server in a halted state. As we increase λ from 30 clients/hour to 100 clients/hour, the average waiting time for the clients increases. This is due to the fact that with a higher number of clients arriving in the system, contention for resources increases. This leads to an increase in the length of the queue which increases the time to serve clients. Now, if we further increase λ to 110 clients/hour, the system reaches a saturation point, where the arrival rate is almost equal to the service rate and the waiting time for clients increases sharply.

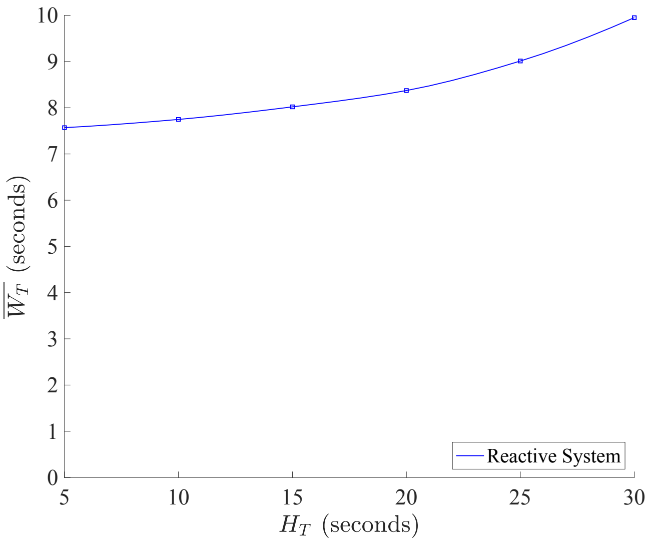


λ	Value
30	5.943
40	6.217
50	6.826
60	7.817
80	10.861
100	17.749
110	29.556

Figure 4.6: Effect of λ on $\overline{W_T}$

4.4.2 Effect of Hold Time on the Mean Waiting Time

When we increase the virtual machine hold time in the system, we observe a monotonic increase and as shown in Figure 4.7. In this system, clients arrive at a rate of 60 clients/hour and the hold time is varied from 5 to 30 seconds. As we increase the hold time, it results in an increase in the waiting time for the clients. This is due to the fact that the more time the clients spend in the system, it increases the load on the server as it is handling various process at the same time, which leads to the higher start time and thus increasing the overall average waiting time for the clients. It is the same behavior as observed in the baseline system. Also, the hold time cannot be increased beyond 30 seconds because at this time, the system is already at its maximum capacity as the default arrival rate is set to 60 clients/hour and deleting a virtual machine takes on average 27 seconds, this makes the service rate 61 clients/hour.



H_T	Value
5	7.569
10	7.747
15	8.019
20	8.372
25	9.011
30	9.499

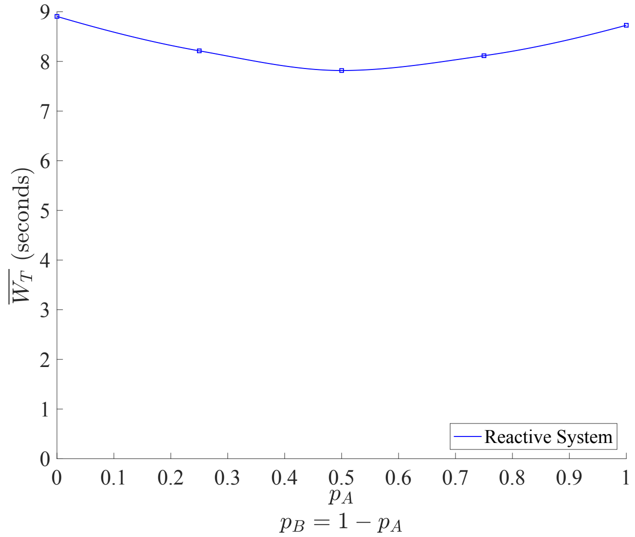
Figure 4.7: Effect of H_T on $\overline{W_T}$

4.4.3 Effect of Probability of Selecting the Type of VM on the Mean Waiting Time

Figure 4.8 presents the impact of the probability of selecting the type of the virtual machine on the mean waiting time for the client. In the case of the reactive system, when the probability of selecting a virtual machine is varied, the average waiting time of clients shows a non-monotonic behavior. At $p_A = 0$, all the clients coming in the system are demanding a virtual machine of type B, but the pool has an equal number of VMs of each type. Thus, once all virtual machines of type B in the pool are allocated, the request starts queuing up and increases the waiting time. Also, at this point, the pre-created virtual machines of type A remains unused in the system. However, when the probability of selecting a virtual machine is equal for both types i.e. $p_A = p_B = 0.5$ the waiting time decreases. The decrease in the waiting time is due to the fact that the pool has both types of virtual machines are pre-created (with 50% ratio for each type of virtual machine). In this case, the clients coming into the system are demanding for both types of virtual machines and therefore, the system can accommodate more clients before requests start queuing up. When we reach the point where $p_A = 1$, all the clients coming in the system are demanding a virtual machine of type A. This situation is the same as the one explained when $p_A = 0$ (i.e. the average waiting time increases).

4.4.4 Effect of Pool Size on the Mean Waiting Time

Figure 4.9 depicts how the average waiting time of clients is affected by varying the pool size. For example, a pool size of 30, there are 15 virtual machines of each type. When increasing the pool size, the average waiting time for clients decreases. The decrease



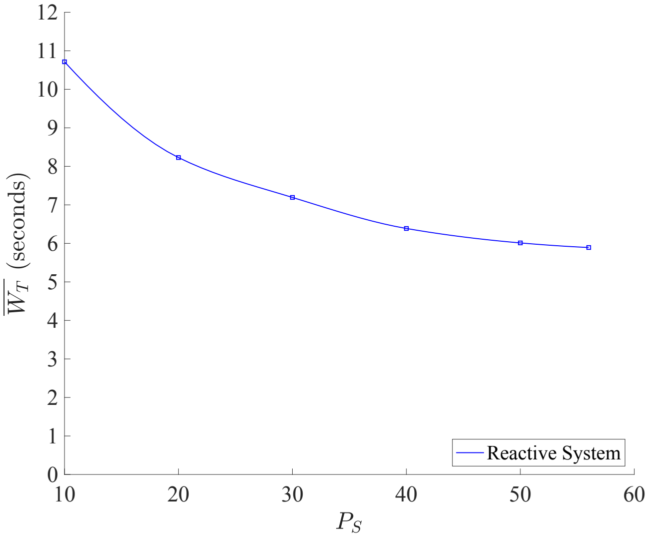
p_A	Value
0	8.906
0.25	8.213
0.5	7.817
0.75	8.115
1	8.728

Figure 4.8: Effect of p_A on $\overline{W}_T$

in the wait time is due to the fact that the number of clients coming in has a constant arrival rate of 60 clients/hour for this experiment. Therefore, increasing the pool size means that more clients will be served right away from a pre-created VM. More precisely, At P_S = 10, there are only 5 virtual machines of each type. Therefore, the queuing starts to occur due to the small pool size which results in an average wait time of 10.714 seconds. When increasing the pool size to 30, more clients can be accommodated before the queuing, therefore the average waiting time decreases. It is important to note that due to memory and storage restrictions of host server used in the experiments, we cannot accommodate more than 56 virtual machines.

4.4.5 Effect of Arrival Rate on the Mean Idle Time

Virtual machines are pre-created and maintained in the pool (in the halted state) until a client requests a virtual machine. Figure 4.10 shows the effect of arrival rate on the



P_S	Value
10	10.714
20	8.228
30	7.193
40	6.387
50	6.012
56	5.893

Figure 4.9: Effect of P_S on $\overline{W}_T$

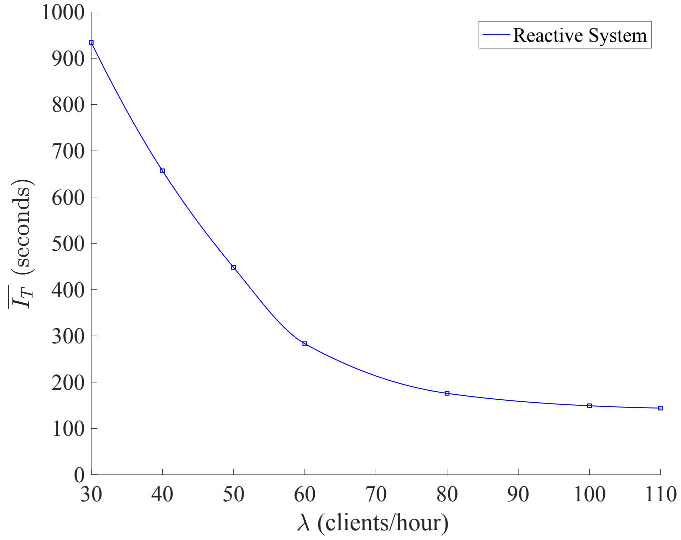
average idle time of the virtual machines. We observe a decreasing behavior because as the number of clients arriving in the system increases, the virtual machines are allocated faster to the users, which means they spend less time in the halted state while maintaining the pool size.

4.5 Experiments for the Proactive System

This section explains the effect of different workload parameters on the average waiting time of the clients and the average idle time of the virtual machines in the proactive system. The simulations were performed for a period of 180 minutes.

4.5.1 Effect of Arrival Rate on the Mean Waiting Time

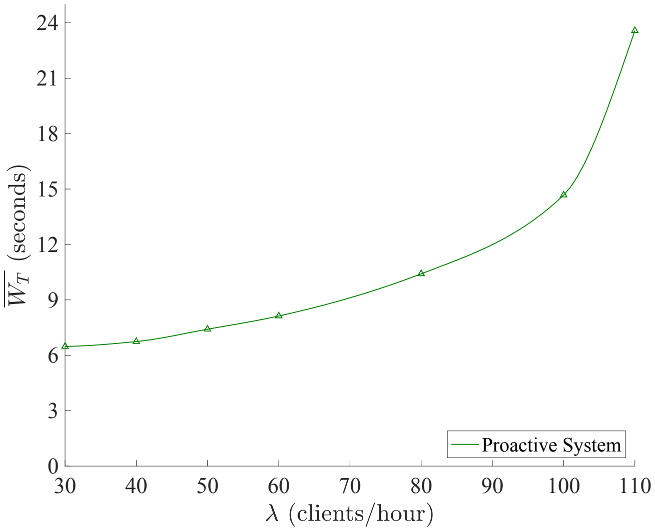
In the case of the proactive system, the waiting time is calculated as the time spent in the queue before the allocation of a virtual machine. Figure 4.11 explains the effect



λ	Value
30	934.037
40	656.857
50	448.119
60	283.222
80	175.884
100	149.016
110	144.334

Figure 4.10: Effect of λ on $\overline{I}_T$

of the arrival rate on the average waiting time of the client. In the proactive system, the pool size is not static, it is predicted based on the previous client requests arriving in the system. To predict the number of virtual machines that would be required for each type, an autoregression model is used (see Section 3.3.2). The prediction is done for every 10 minutes by learning the behavior of the clients from the past data. As we increase the arrival rate, the client's average waiting time increases. At the point where $\lambda = 30$ clients/hour, the average waiting time is 6.447 seconds. If we double the arrival rate to 60 clients/hour there is only a difference of 1.68 seconds. The waiting time does not increase significantly until the value of λ becomes 100. This is due to the reason that as we increase the arrival rate, the average number of clients in the system also increases, which means that the model has more data to train on and thus gives a prediction value with a lower error (i.e. a better prediction). These forecast values are the number of clients that might be coming in the system in next 10 minutes for each type of the virtual machine. At λ = 110, the system saturates and thus, the waiting time increases rapidly.

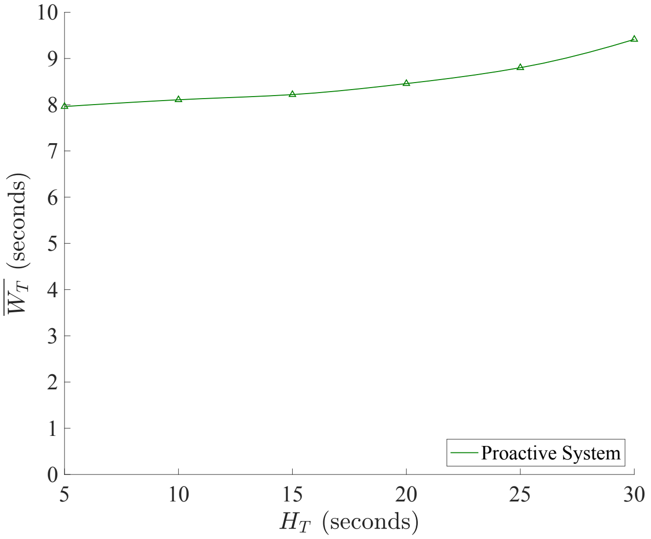


λ	Value
30	6.447
40	6.478
50	7.412
60	8.127
80	10.411
100	14.673
110	23.574

Figure 4.11: Effect of λ on $\overline{W}_T$

4.5.2 Effect of Hold Time on the Mean Waiting Time

The proactive system also shows an increase in the average waiting time for the clients with respect to the virtual machine hold time. In this system, the clients arrive at a default rate of 60 clients/hour and the hold time is varied from 5 to 30 seconds. As we increase the hold time of a virtual machine, it results in the increment of the waiting time for the clients. This is due to the reason that if clients spend more time in the system, it increases the load on the server as it is handling various processes at the same time, which leads to the higher start time of a virtual machine and thus increasing the overall average waiting time of clients.

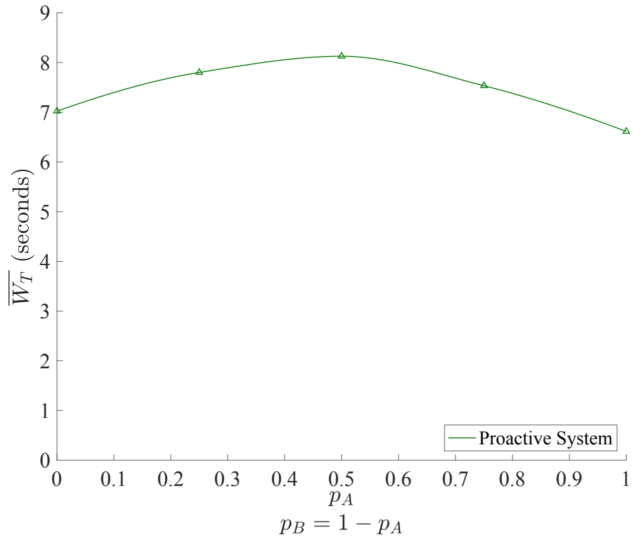


H_T	Value
5	7.961
10	8.107
15	8.219
20	8.457
25	8.801
30	9.413

Figure 4.12: Effect of H_T on $\overline{W}_T$

4.5.3 Effect of Probability of Selecting the Type of VM on the Mean Waiting Time

Figure 4.13 shows how the proactive system behaves when the probability of the type of virtual machine selection is varied. At point when $\mathrm{p_A}$ is 0 (case 1), all the clients are requesting a virtual machine of type B. Now, the predictor will forecast the values of both types of virtual machines but as all the past values of type A are 0, the predictor will forecast 0 for type A machines which means that there is no error in prediction. However, while predicting the number of type B machines, there is a significant error as the model has fewer data to train. Thus the waiting time is only dependent on the error percentage of type B virtual machines and we obtain a value of 7.023 seconds. When the probability of selecting the type of the virtual machine is 0.5 for each type (case 2). The error in the prediction value for both types of virtual machines, therefore, the waiting time increases. At p_A = 1, all the clients coming in the system are demanding the virtual machines of type A, so similar to case 1, the average waiting time of clients decreases.

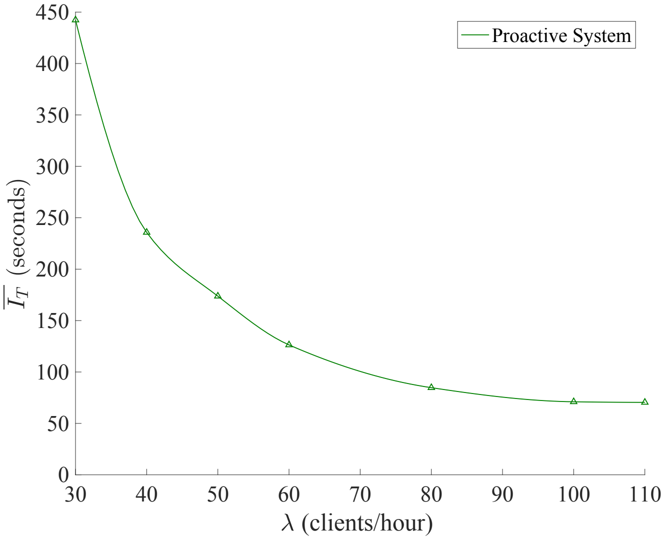


p_x	Value
0	7.023
0.25	7.801
0.5	8.127
0.75	7.533
1	6.614

Figure 4.13: Effect of p_x on $\overline{W}_T$

4.5.4 Effect of Arrival Time on the Mean Idle Time

The virtual machines are dynamically created based on the past history of the clients arriving in the system and the type of service they are demanding for. We observed a decrease in the average idle time of virtual machines with an increase in the arrival rate. This behavior is shown in the Figure 4.14. With the arrival rate of 30 clients/hour, we have a higher average idle time for all the virtual machines that were allocated to the clients. This is because, with a lower arrival rate, the predictor has fewer data to train. So when the predictor forecasts higher threshold values, more virtual machines are created and left idle in the pool. This increases the idle time of the virtual machines. When predictor forecast lower threshold values it decreases the idle time of the virtual machine. But as we increase the arrival rate of clients, the predictor has more data to learn the behavior of requests and forecasts more accurate threshold values. This reduces the number of virtual machines that will be created and stacked in the pool in the halted state, which decreases the overall average idle time.



λ	Value
30	442.343
40	235.768
50	173.753
60	126.342
80	84.714
100	71.026
110	70.453

Figure 4.14: Effect of λ on $\overline{I}_T$

4.6 Comparison of the Systems

In the previous sections, we studied the effect of different workload parameters on the performance of all the three systems individually. This section provides a comparison of the three systems. We will also discuss the suitability of a system for a given set of workload matrices. The run length for all the experiments is fixed to 180 minutes.

4.6.1 Effect of Arrival Rate on the Mean Waiting Time

Figure 4.15 compares the performance of all the three systems. As seen earlier, as we increase the arrival rate, the waiting time for the clients is increasing. But, the point to note here is that the waiting time increases more sharply for the baseline system. This is due to fact that the clients coming in the baseline system have to wait for a VM until it is created and then started. The baseline system reaches its saturation point when λ reaches 68 clients/hour, whereas in case of reactive and proactive systems the saturation point is attained at higher values of the arrival rate (around 110 clients/hour). The reason for this is the fact that for the proactive and reactive systems, the virtual machines are created a-priori and maintained in a pool. This means that the system is ready with virtual machines before the client arrives in the system. Therefore, the waiting time is only the time spent in the queue. Further, if we compare the reactive system with the proactive system, we observe that for λ lower than 70 clients/hour, the reactive system has less waiting time than the proactive system and for λ above 70 client/hour, the proactive system is better and has a lower waiting time. This is due to the fact that in the case of the proactive system, as the number of clients arriving in the system increases, the data on which the model can train itself increases, which tends to decrease the percentage of error in the predicted values. The proactive system is dynamic in nature and prepares itself by adjusting the threshold values as required. Thus, with a higher prediction accuracy, it performs better than the reactive system. The proactive system creates or destroys the virtual machines based on the predicted threshold values to make the pool ready to accommodate the incoming clients. As a result, the number of clients waiting in the queue decreases, which gives us an overall average waiting time that is lower than a reactive system. In the case of the reactive system, the pool size remains the same and the number of virtual machines created a-prior is fixed. Due to this reason, the waiting time is lesser for smaller values of λ. However, when increasing λ, the queuing time increases because virtual machines need to be created to maintain the pool size and creating a virtual machine requires time which increases the time spent in the queue by the client, and thus leads to an increased overall average waiting time. Further, the difference between the reactive and proactive systems increases, for λ = 110 clients/hour the average waiting time achieved is 23.57 seconds for the proactive system which is 25.37% lower than the reactive system.

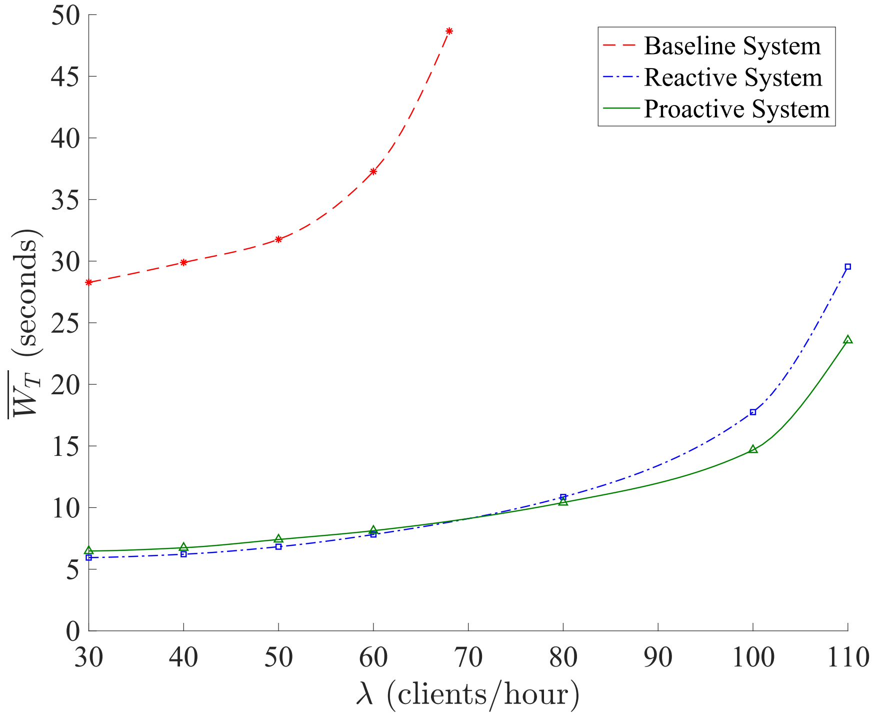


Figure 4.15: Effect of λ on $\overline{W}_T$

4.6.2 Effect of Hold Time on the Mean Waiting Time

The impact of the average VM hold time on the waiting time is presented in the Figure 4.16. Here, the number of clients arriving in the system is fixed at its default value: 60 clients/hour. We observe an increase in the average waiting time for all the three systems. The reason for this increase is that starting a virtual machine requires some time and this time is directly proportional to the load on the server.

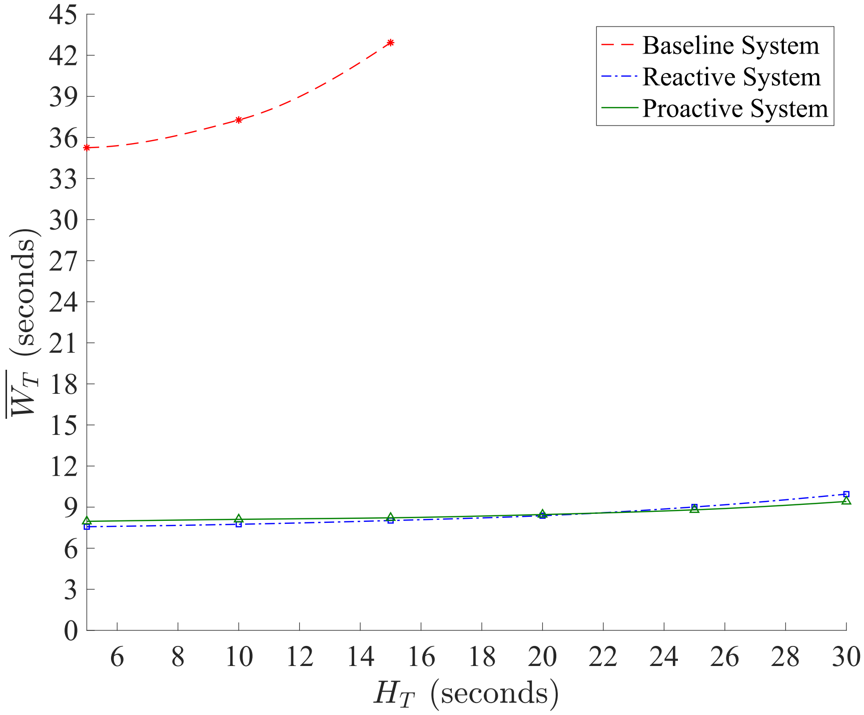


Figure 4.16: Effect of H_T on $\overline{W}_T$

In the case of the baseline system, the load on the server is due to factors like the time required for creating virtual machines, running virtual machines on the server, and destroying virtual machines. Thus, when H_T is varied from 5 to 15 seconds, the load on the server increases which further increases the virtual machine start time, that increases the overall average waiting for the clients. In the case of the reactive system, the increase in the average waiting time of the client with an increase in the virtual machine hold time is very small. In the reactive system, the load on the server is due to following reasons: the number of virtual machines in the pool and the number of virtual machines which are in the active state (i.e. used by the client). Now, as the H_T increases, the amount of time clients spend on the server increases. This means that a higher number of virtual machines are active and for a longer duration. Thus, the virtual machine's start time increases which result in an increase in the overall waiting time of the clients. The reactive and proactive systems could handle higher H_T when compared to the baseline system, the reason being the waiting time of the clients does not include the creation time. The proactive and reactive system's results produce comparable values of the average waiting time for a given arrival rate .

4.6.3 Effect of Probability of Selecting the Type of VM on the Mean Waiting Time

The impact of the probability of selecting a specific type of virtual on the average waiting time of the client is displayed in Figure 4.17. While evaluating this graph, we observed that among the three systems, the baseline system has the highest average waiting time. When the value of p_A the number of clients coming in the system are all demanding for a virtual machine of type B. As we move towards $p_A = 1$, the number of clients requesting virtual machine of type A is increasing and when the value becomes 1, all the clients are demanding virtual machine of type A. Because the size of virtual machine of type A is smaller than the size of a virtual machine of type B (256 MB vs 512 MB) virtual machines of type A takes less time to create and start. As a result, the overall average waiting time decreases as the number of clients demanding virtual machine of type A increases. When comparing the reactive and proactive systems at a point where the probability of selecting a virtual machine is half for each type, the system shows an increase in the average waiting time of the clients in comparison to the reactive system. This behavior is only shown in this case where the arrival rate is fixed at 60 clients/hour and the hold time is held at 10 seconds. But, when we increase the arrival rate, the average waiting time decreases and the proactive system becomes the best option among the three systems. The reason is that the more historical data the system can use, the better the prediction.

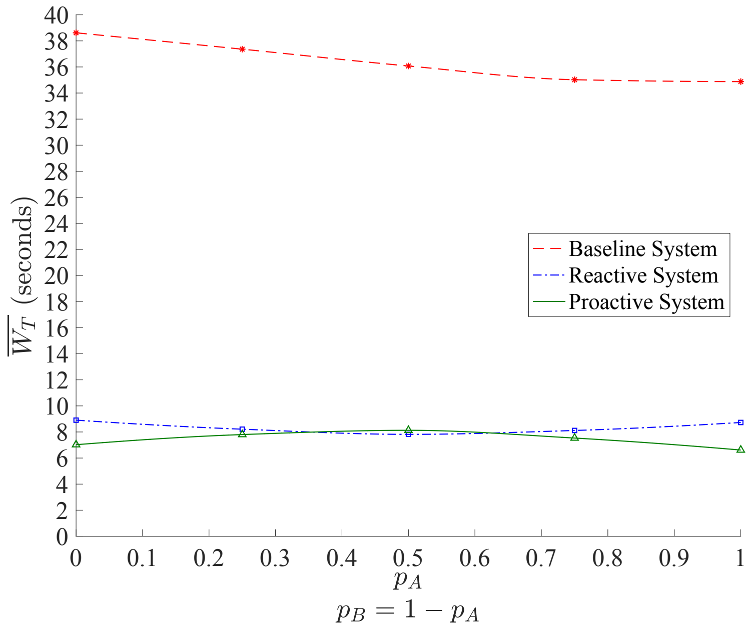


Figure 4.17: Effect of p_x on $\overline{W}_T$

4.6.4 Effect of Arrival Rate on the Mean Idle Time

The effect of the arrival rate on the mean idle time of virtual machines is shown in Figure 4.18. As, discussed earlier, both the reactive and proactive systems show a decrease of the idle time when the arrival rate is increased from 30 clients/hour to 110 clients/hour. When comparing the reactive and proactive systems, the average idle time of a virtual machine is lower in the proactive system. The reason for the difference in the performance of the two systems is that the proactive system has a dynamic pool thus, only the required number of virtual machines are created a-priori and this number is forecasted by the predictor based on the historical data. Further, the decrease could be justified, by saying that when the number of clients arriving in the system increases, the predictor has more data to train and gives a more accurate prediction. As the accuracy of the prediction increases, the pool logic reconfigures the system to accommodate the incoming clients which decrease the idle time for a virtual machine.

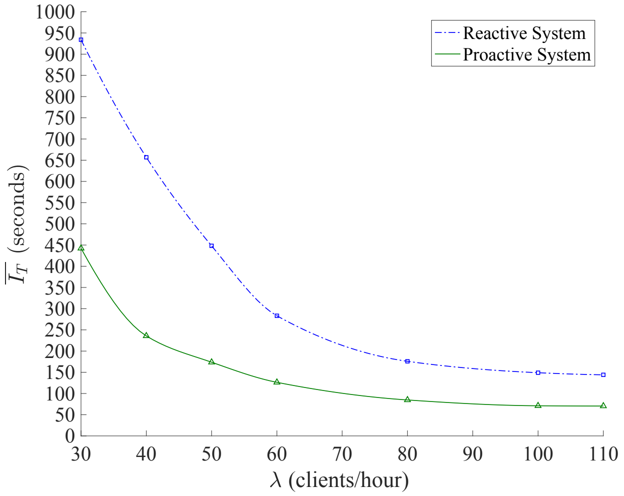


Figure 4.18: Effect of λ on $\overline{I}_T$

Chapter 5

Conclusions & Future Work

This chapter first provides a summary of the proposed adaptive system for allocating virtual machines to clients to provide secure remote access via a cloud environment. The goals of the system are to reduce user waiting time and reduce the amount of idle resources to improve the overall performance of the system. Next, it describes the salient features of the three systems designed in order to achieve the thesis goals. Finally, it highlights a few ideas that could be added in the future to further improve the system.

5.1 Summary & Contributions

As companies shift from desktop applications to cloud-based SaaS applications(such as vKey), the competition for the end-user experience by cloud providers offering similar services grows. In order to survive in such a competitive market, cloud-based companies must achieve a good quality of service for their users along with ensuring the security of the confidential data. However, meeting the user requirements, reducing user waiting time and reducing the amount of idle resources in a cost effective manner is a challenging task as the rate of arrival for user requests varies over time. This is the challenge we were trying to solve in this collaborative research project with DLS.

The objectives of this thesis research that is directed at addressing these issues were discussed in Chapter 1. Since clients arriving in the system do not follow a fixed pattern, the company could end up with a monetary loss if the amount of idle resources is not controlled. Also, if the system is not provisioned to accommodate the client workload, the company might face a loss of reputation. In this thesis, we proposed an adaptive system to dynamically allocate virtual machines to users based on the prediction of client arrivals using autoregression. The system can predict the future needs based on the history of the users and reconfigure the system in advance to accommodate the client workload. We refer to that system as the proactive system and a real prototype for this system was built and tested. Results show that the proactive system can reduce waiting time by 78.19% compared to a baseline system and can reduce the idle number of resources by 55.39% compared to a reactive system. Finally, the prototype was transfered to DLS Technology so that they can deploy it in a real work environment.

In the first system (referred to as the baseline system), we used a straightforward approach to allocate virtual machines to clients. Once a client comes into the system requesting a virtual machine, the algorithm simply creates a virtual machine from a given template and allocates it to the client. As previously mentioned, creating a virtual machine takes a finite amount of time and as a result, clients arriving within a short period of time start to queue up in the system. Various scenarios were used to calculate the waiting time for different arrival rates in order to attain the results stated in Chapter 4. Results revealed that, although simple, this system is not the best way to address the requirements of the clients as their waiting time increases greatly. This led to the devising of the reactive and proactive systems that gave rise to a significant improvement in performance. A short discussion on the comparison of the proactive and reactive systems is presented next.

Comparison between the proactive and the reactive systems

The following observations are based on the results of the experiments described in the thesis.

- At higher arrival rates, the average waiting time of clients for the proactive system is significantly lower than that achieved on the reactive system.
- The average idle time achieved with the proactive system for a given arrival rate is smaller in comparison to that achieved in the reactive system.
- The threshold values for the number of virtual machines of type A and type B used in the reactive system are tuning parameters that need to be determined by the system administrator. No such parameter tuning is required with the proactive system.

5.1.1 Digest of the Contributions

This subsection summarizes the major findings of the thesis and related results.

- To address the problems in the baseline system, a different approach was designed in which a pool of virtual machines is maintained in a halted state such that when a client comes in with a request, a virtual machine is usually available to satisfy the requirements. As discussed in Chapter 4, this approach resulted in a lower waiting time for the clients under various scenarios. However, from the results, we also concluded that maintaining a fixed size pool of virtual machines leads to a waste of resources and idle resources can further be linked to energy consumption, cost of operation, etc. which might incur a monetary loss.

- To overcome the problem of idle resources, we proposed a proactive approach for dynamic allocation of virtual machines based on the historical data of the client arrivals. The approach analyzes the client behavior by using autoregression. It predicts the number of clients that might be coming into the system during a specific period of time. Then, it reconfigures the system and maintains only the predicted number of virtual machines in the pool and as a result, the amount of idle resources were reduced. Further, we also evaluated the effect of this approach on the waiting time of the clients. The simulation results discussed in Chapter 4 showed that this approach proves to be best as it reduces the average waiting time and the amount of idle resources in comparison to the reactive system thus benefiting both users and the company.

5.2 Future Work

To the best of our knowledge, this system is one of a kind and if DLS Technology decides to deploy the system in a production environment the system might end up being registered as a patent. But before that, here are few ideas that could be used in order to further improve the system.

- The predictions made in the proactive systems could be used to find the ratio of the different clients that will join the system. This ratio then can be used to create a proportional number of virtual machines based on the free space available to fill the hard disk. Since the VMs that are waiting to be assigned are in halted mode, this would not consume more resources (CPU memory) and could provide slightly more flexible since it could provide extra VMs in case predictions were not accurate.

- A live migration of virtual machines could also be incorporated based on the storage specification. Migrating the running virtual machines could allow the company to shutdown servers that are partially used which could eventually save energy, therefore decreasing the overhead to the company.
- In the proactive system, the predictions could me made more accurate by using a different model to analyze the workload component in terms of time series. For example, neural network models are powerful enough to learn the most important past behaviors and understand whether or not those past behaviors are important features in making future predictions. It is like a model which has its own memory and which can behave like an intelligent human in making decisions.

Appendices

Appendix A

Accuracy of Measurement: Confidence Intervals

The accuracy of the measurements made during the experiments is captured in the confidence intervals associated with the average waiting time of clients. Three sample graphs are presented in figures A.1, A.2, A.3. For all the points in the three graphs, the confidence interval for the average waiting time of the users was found to be less than 7.18% of the mean at a confidence level of 95%. Since there was a large time associated with running the experiments, the confidence intervals could only be determined for this sample set of the three graphs.

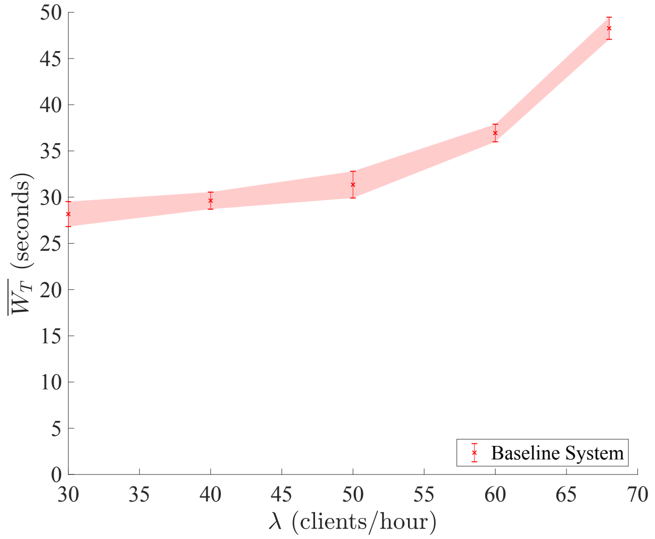


Figure A.1: Confidence intervals for the baseline system

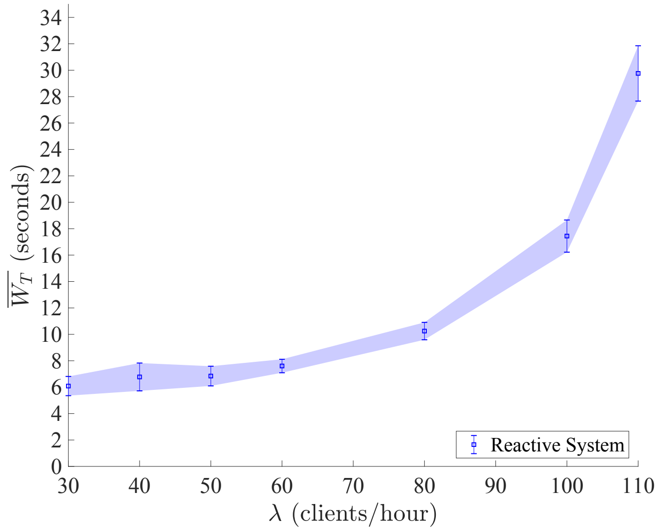


Figure A.2: Confidence intervals for the reactive system

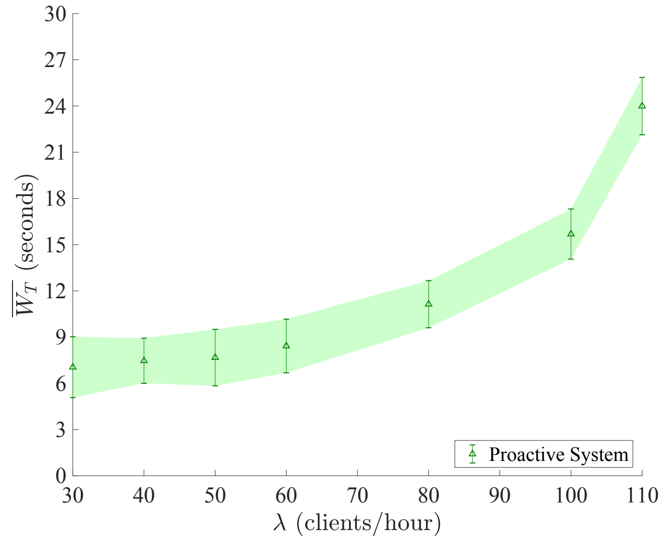


Figure A.3: Confidence intervals for the proactive system

Appendix B

Accuracy of Measurement: Predicted Values

To evaluate the effect of the arrival rate on the accuracy of the predicted values, we ran three separate experiments with different arrival rate (50, 100 and 500 clients per minute). These experiments ran for a total of 60 minutes with clients arriving according to a Poisson process. The first 50 minutes were used to train the algorithm and the last 10 minutes were used to compare the predicted values to the actual values.

In figures B.1, B.2 and B.3, the x-axis represents the simulation time and the y-axis represents the number of client request that was received during a specific time interval. The red points correspond to the actual values and blue points correspond to the predicted values. As can be seen, the predicted values seem to be close to the actual values. However, with an increase in the arrival rate, the predictions become more accurate. This is due to the fact that with higher arrival rates, the algorithm has more data to train with.

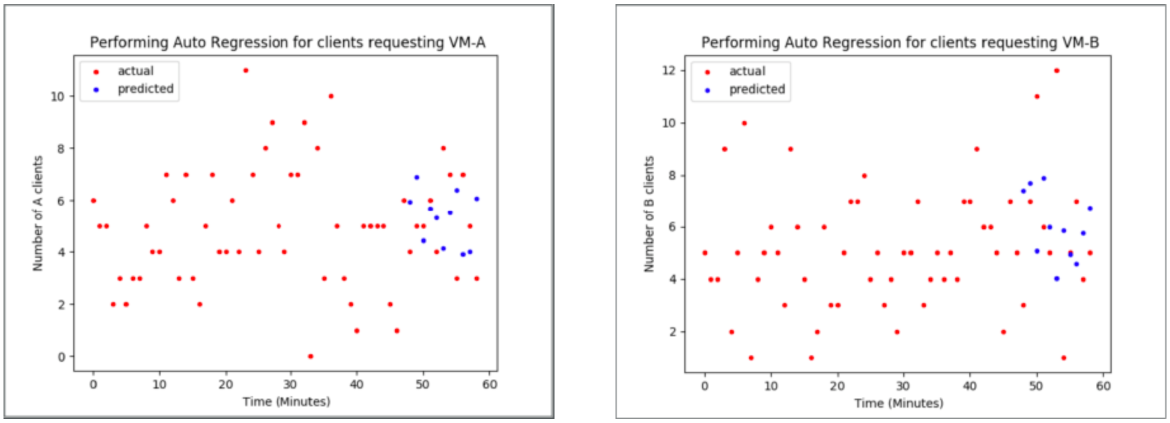


Figure B.1: Actual values vs predicted values for $\lambda = 50$ clients/minute

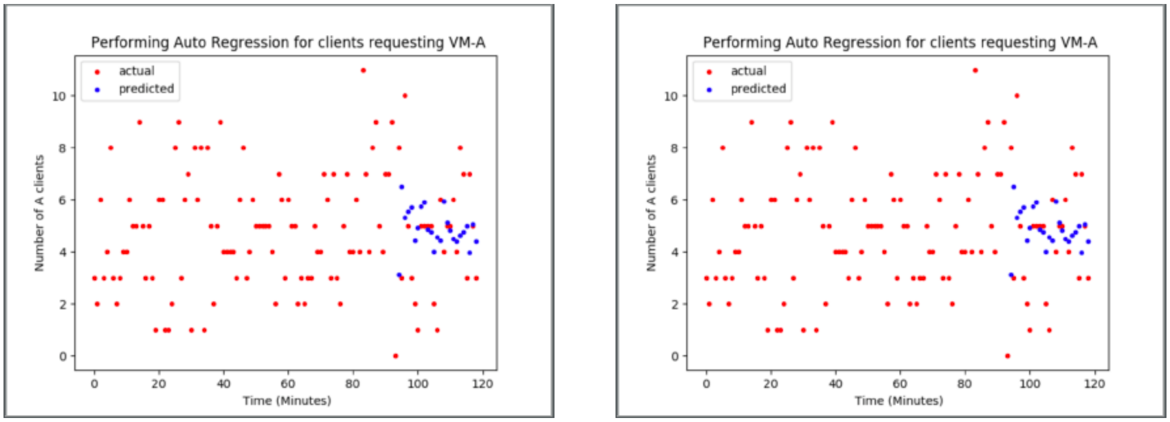


Figure B.2: Actual values vs predicted values for $\lambda = 100$ clients/minute

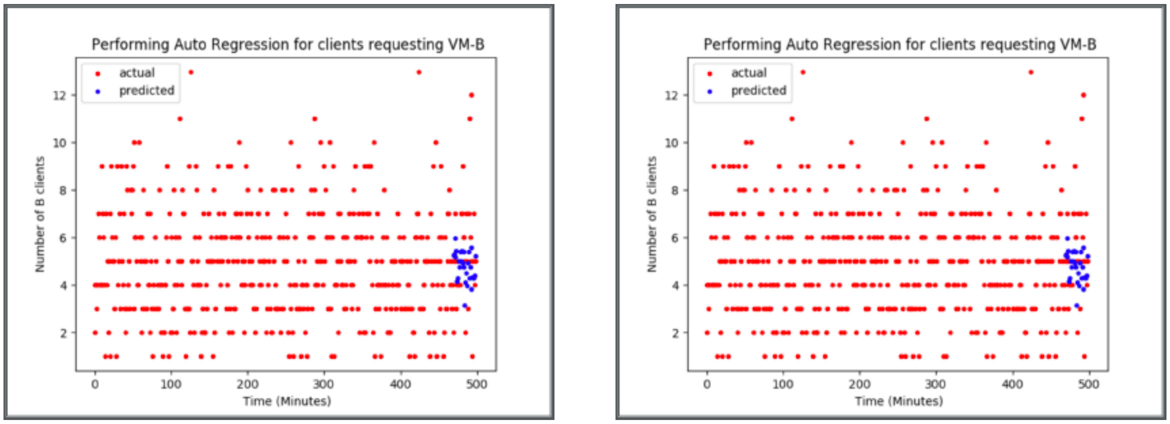


Figure B.3: Actual values vs predicted values for λ = 500 clients/minute

References

[1] DLS Technology, "vKey Technologies." 2018. [Online]. Available: `http://www.dlstech.com/products`. [Accessed: 12-Jun-2018].

[2] J. Niemi, *Empowering IT Solutions with Server Virtualization*. PhD thesis, Turku University of Applied Sciences, Finland.

[3] G. Ahmed, *Implementing Citrix XenServer Quickstarter*. Packt Publishing Ltd, 2013.

[4] C. Takemura and L. S. Crawford, *The Book of Xen: A Practical Guide for the System Administrator*. No Starch Press, 2010.

[5] T. Cerling and J. L. Buller, *Mastering Microsoft Virtualization*. John Wiley & Sons, 2011.

[6] W. Von Hagen, *Professional Xen Virtualization*. John Wiley & Sons, 2008.

[7] C. Pettey, "Cloud Computing Will Be As Influential As E-business." 2018. [Online]. Available: `https://www.gartner.com/newsroom/id/707508`. [Accessed: 08-Jan-2018].

[8] W.-T. Tsai, X. Sun, and J. Balasooriya, "Service Oriented Cloud Computing Architecture," in *Seventh IEEE International Conference on New Generations in Information Technology (ITNG)*, pp. 684–689, 2010.

[9] S. Ray, "7 Types of Regression Techniques you should know." 2017. [Online]. Available: https://www.analyticsvidhya.com. [Accessed: 03-Feb-2018].

[10] A. Kivity, Y. Kamay, D. Laor, U. Lublin, and A. Liguori, "KVM: The Linux virtual Machine Monitor," in *The Linux Symposium*, vol. 1, pp. 225–230, 2007.

[11] F. Bellard, "QEMU, A Fast and Portable Dynamic Translator," in *Annual Conference on USENIX Annual Technical Conference*, ATEC '05, pp. 41–41, USENIX Association, 2005.

[12] T. Abels, P. Dhawan, and B. Chandrasekaran, "An Overview of Xen Virtualization," in *Dell Power Solutions*, vol. 8, pp. 109–111, 2005.

[13] T. Mackey, "XenServer: Core Architecture and Critical Components." 2018. [Online]. Available: https://www.oreilly.com/learning. [Accessed: 11-Jan-2018].

[14] Citrix, "Xenserver-7-0-Management-API-Guide." 2018. [Online]. Available: https://docs.citrix.com/content/dam/docs/en-us/xenserver. [Accessed: 03-Feb-2018].

[15] S. Ray, "Essentials of Machine Learning Algorithms (with Python and R Codes)," *Analytics Vidhya*, vol. 10, no. 08, 2015.

[16] R. Adhikari and R. Agrawal, "An Introductory Study on Time Series Modeling and Forecasting," *arXiv:1302.6613*, 2013.

[17] S. S. Priya and L. Gupta, "Predicting the Future in Time Series using Auto Regressive Linear Regression Modeling," in *Twelfth IEEE International Conference on Wireless and Optical Communications Networks (WOCN)*, pp. 1–4, 2015.

[18] A. Beloglazov and R. Buyya, "Energy Efficient Allocation of Virtual Machines in Cloud Data Centers," in *10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing (CCGrid)*, pp. 577–578, 2010.

[19] Z. Mann, "Allocation of Virtual Machines in Cloud Data Centers—A Survey of Problem Models and Optimization Algorithms," in *ACM Computing Surveys (CSUR)*, vol. 48, p. 11, 2015.

[20] M. M. Nejad, L. Mashayekhy, and D. Grosu, "Truthful Greedy Mechanisms for Dynamic Virtual Machine Provisioning and Allocation in Clouds," in *IEEE Transactions on Parallel and Distributed Systems*, vol. 26, pp. 594–603, 2015.

[21] S. Zaman and D. Grosu, "Combinatorial Auction-based Allocation of Virtual Machine Instances in Clouds," in *Journal of Parallel and Distributed Computing*, vol. 73, pp. 495–508, Elsevier, 2013.

[22] R. A. Gagliano, M. D. Fraser, and M. E. Schaefer, "Auction Allocation of Computing Resources," in *Communications of the ACM*, vol. 38, pp. 88–102, 1995.

[23] J. Gomoluch and M. Schroeder, "Performance Evaluation of Market-based Resource Allocation for Grid Computing," in *Practice and Experience in Concurrency and Computation*, vol. 16, pp. 469–475, Wiley Online Library, 2004.

[24] D. Lehmann, L. I. Oćallaghan, and Y. Shoham, "Truth Revelation in Approximately Efficient Combinatorial Auctions," in *Journal of the ACM (JACM)*, vol. 49, pp. 577–602, 2002.

[25] R. Wolski, J. S. Plank, J. Brevik, and T. Bryan, "Analyzing Market-based Resource Allocation Strategies for the Computational Grid," in *The International Journal of High Performance Computing Applications*, vol. 15, pp. 258–281, Sage Publications Sage CA: Thousand Oaks, CA, 2001.

[26] Q. Zhu and G. Agrawal, "Resource Provisioning with Budget Constraints for Adaptive Applications in Cloud Environments," in *19th ACM International Symposium on High Performance Distributed Computing*, pp. 304–307, ACM, 2010.

[27] N. Bonvin, T. G. Papaioannou, and K. Aberer, "Autonomic SLA-Driven Provisioning for Cloud Applications," in *11th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, pp. 434–443, 2011.

[28] Z. Tang, Y. Mo, K. Li, and K. Li, "Dynamic Forecast Scheduling Algorithm for Virtual Machine Placement in Cloud Computing Environment," in *The Journal of Supercomputing*, vol. 70, pp. 1279–1296, Springer, 2014.

[29] S. B. Shaw and A. K. Singh, "Use of Proactive and Reactive Hotspot Detection Technique to Reduce the Number of Virtual Machine Migration and Energy Consumption in Cloud Data Center," in *Computers & Electrical Engineering*, vol. 47, pp. 241–254, Elsevier, 2015.

[30] N. Roy, A. Dubey, and A. Gokhale, "Efficient Autoscaling in the Cloud using Predictive Models for Workload Forecasting," in *IEEE International Conference on Cloud Computing (CLOUD)*, pp. 500–507, 2011.

[31] V. Debusschere, S. Bacha, *et al.*, "Hourly Server Workload Forecasting up to 168

hours Ahead using Seasonal ARIMA Model," in *IEEE International Conference on Industrial Technology (ICIT)*, pp. 1127–1131, 2012.

[32] R. N. Calheiros, E. Masoumi, R. Ranjan, and R. Buyya, "Workload Prediction using ARIMA Model and its Impact on Cloud Applications," in *IEEE Transactions on Cloud Computing*, vol. 3, pp. 449–458, 2015.

[33] K. R. Lee, "Impacts of Information Technology on Society in the new Century," in *Route de Chavannes C*, vol. 27, 2002.

[34] B. Clark *et al.*, "Xen and the Art of Repeated Research," in *FREENIX Track, USENIX Annual Technical Conference*, pp. 135–144, 2004.

[35] I. E. Sutherland, "A Futures Market in Computer Time," in *Communications of the ACM*, vol. 11, pp. 449–451, 1968.

[36] R. Buyya, R. Ranjan, and R. N. Calheiros, "Intercloud: Utility-oriented Federation of Cloud Computing Environments for Scaling of Application Services," in *International Conference on Algorithms and Architectures for Parallel Processing*, pp. 13–31, Springer, 2010.

[37] R. Jansen and P. R. Brenner, "Energy Efficient Virtual Machine Allocation in the Cloud," in *IEEE International Green Computing Conference and Workshops (IGCC)*, pp. 1–8, 2011.

[38] H. Goudarzi and M. Pedram, "Energy-efficient Virtual Machine Replication and Placement in a Cloud Computing System," in *5th IEEE International Conference on Cloud Computing (CLOUD)*, pp. 750–757, 2012.

[39] P. Raycroft, R. Jansen, M. Jarus, and P. R. Brenner, "Performance Bounded Energy Efficient Virtual Machine Allocation in the Global Cloud," in *Sustainable Computing on Informatics and Systems*, vol. 4, pp. 1–9, Elsevier, 2014.

[40] A. Beloglazov and R. Buyya, "Optimal Online Deterministic Algorithms and Adaptive Heuristics for Energy and Performance Efficient Dynamic Consolidation of Virtual Machines in Cloud Data Centers," in *Practice and Experience in Concurrency and Computation*, vol. 24, pp. 1397–1420, Wiley Online Library, 2012.

[41] S. B. Shaw and A. K. Singh, "Use of Proactive and Reactive Hotspot Detection Technique to Reduce the Number of Virtual Machine Migration and Energy Consumption in Cloud Data Center," in *Computers & Electrical Engineering*, vol. 47, pp. 241–254, Elsevier, 2015.

[42] E. Caron, F. Desprez, and A. Muresan, "Forecasting for grid and cloud computing on-demand resources based on pattern matching," in *IEEE Second International Conference on Cloud Computing Technology and Science (CloudCom)*, pp. 456–463, 2010.

[43] C. Chatfield, *An Introduction to the Analysis of Time Series*. CRC press, 2016.