"buna": Automated Administration tool for Hosts & Virtual Machines under Libvirt Environment

Master Thesis by

Solomon A. Habtu

Submitted to Network and System Administration Program.

December 18, 2011
"buna": Automated Administration tool for Hosts & Virtual Machines under Libvirt Environment

Master Thesis by
Solomon A. Habtu
Submitted to Network and System Administration Program.

December 18, 2011
Abstract

Virtualization is very common and widely used in many data-centers nowadays. Easy management and administration of virtual machines and hosts is a crucial part of Virtualization. Currently there are automated virtual machine management systems that are in use. But the tools lack consideration of the needs of organizations with dynamically changing virtual machine environment, like Universities and Colleges. They are also limited to specific virtualization platforms and don't have host management capability. In this thesis an entirely new approach of virtual machine management based on mathematics Set-theory, Logic and Statistics is suggested. This approach provides a new virtual machine naming method that creates relationships between virtual machines. This makes the virtual machine and Host management automation simple, flexible and powerful.

The suggested approach is implemented as a tool named buna using Perl programming language and libvirt management tools: virsh, virt-clone and virt-top. Buna has two major parts: automation and plugin part. The automation part enables actions like create, start/stop and delete virtual machines; upgrade CPU and Memory and attach/detach devices on mass number of virtual machines. The plugin part controls host resources and status of virtual machines based on rules, where rules have conditions and actions those are set by system administrators. This avoids the need to perform routine tasks for administrators and it makes the virtual machine and host management easy and effective. In addition, critical system level tasks like load-balancing and power-saving can be done by giving organized rules.

buna is tested on an environment with three hosts running KVM/QEMU virtualization system. It successfully performed the automatic virtual machine management actions between the hosts. It’s also given some rules to test its ability to perform actions based on resource usages of virtual machines.
Contents

1 Introduction 11
  1.1 Motivation .................................................. 11
  1.2 Problem Statement ......................................... 12
  1.3 Scope and Limitations .................................... 12
  1.4 Approach ...................................................... 13
    1.4.1 Automation Part ....................................... 13
    1.4.2 Plugin Part ............................................ 13
  1.5 Research Goals ............................................. 14
  1.6 Thesis Structure .......................................... 15

2 Background and Literature Review 16
  2.1 Virtualization ............................................. 16
    2.1.1 Xen and KVM Overview ................................. 16
      2.1.1.1 Architecture .................................. 17
      2.1.1.2 Compatibility ................................ 18
      2.1.1.3 Performance .................................. 19
  2.2 Libvirt Virtualization Modes ............................. 20
    2.2.1 Single Hypervisor .................................... 20
    2.2.2 Multi Hypervisors ................................... 20
    2.2.3 Multi Host Mode ..................................... 21
    2.2.4 When to use Libvirt? ................................. 21
  2.3 Virtual Machine Management Automation Tools .......... 22
    2.3.1 MLN Manager (Linux open source) .................... 23
      2.3.1.1 Functions ..................................... 23
      2.3.1.2 Architecture .................................. 24
      2.3.1.3 How to use MLN? ................................. 24
      2.3.1.3.1 Creating Project Files: ...................... 24
      2.3.1.3.2 Building VM’s & Networks: ................... 26
      2.3.1.3.3 Managing Projects: .......................... 27
    2.3.1.4 Adding Plugin ..................................... 27
    2.3.2 RHEV Manager (Red Hat commercial) ................ 27
      2.3.2.1 Functions ..................................... 27
      2.3.2.2 Architecture .................................. 28
      2.3.2.3 How to Use RHEV? ............................... 28
3 Libvirt Software (Tool-kit) 31

3.1 Overview ................................................. 31
3.2 Installation ................................................. 31
3.3 Connecting to Libvirt API ............................ 32
3.4 Firewall and Network Filtering: ......................... 32
3.5 Limitation of APIs ........................................ 33
3.6 Security and Stability ................................... 33
3.7 Libvirt Architecture ...................................... 34
  3.7.1 Application Programming Interface (API) ............ 34
  3.7.2 Hypervisors ........................................... 35
  3.7.3 Network .............................................. 35
  3.7.4 Storage ............................................... 36
3.8 Libvirt XML Object Definitions ....................... 37
  3.8.1 Domain .............................................. 37
    3.8.1.1 General Definitions ............................. 37
    3.8.1.2 CPU ........................................... 37
    3.8.1.3 Memory ......................................... 38
    3.8.1.4 Life Cycle Controls ............................ 38
    3.8.1.5 Device and File System ......................... 38
    3.8.1.6 Network Interfaces ............................. 39
    3.8.1.7 Other Definitions ............................... 40
  3.8.2 Network .............................................. 40
  3.8.3 Storage ............................................... 40
3.9 Libvirt Virtual Machine Managing Tools .............. 40
  3.9.1 Virsh (command line) ................................. 41
  3.9.2 virt-clone (command line) ........................... 42
  3.9.3 virt-install (command line) ......................... 42
  3.9.4 virt-(top, what, df) ................................ 43
  3.9.5 virt-manager (GUI/command manager) ................ 44
  3.9.6 Guestfish: interactive manager ...................... 44
3.10 Drawbacks of Existing Management Tools ............. 45
  3.10.1 Automation tools .................................... 45
  3.10.2 Libvirt Management Tools ......................... 46

4 "buna": Methodology 47

4.1 Automation Part .......................................... 47
  4.1.1 Creating Name Based Relationship Between VMs ........ 48
  4.1.2 Generating Meaningful Names .......................... 49
  4.1.3 "buna" Naming Convention ............................. 50
4.2 Plugin Part ............................................... 52
CONTENTS

4.2.1 Methods of Managing Hosts ...................................... 52
4.2.2 Skeleton Program .................................................. 52

5 "buna": Design .......................................................... 56
5.1 Context Diagram ....................................................... 56
5.1.1 Input Options ....................................................... 57
5.1.2 Configuration File (buna.conf) .................................. 60
5.1.3 Rules File .......................................................... 61
5.2 High Level Structure ................................................... 61
5.2.1 Option-Manager ..................................................... 61
5.2.2 Global-Variable-Updater .......................................... 63
5.2.3 Name-Generator ..................................................... 64
5.2.4 Name-Filter ........................................................ 67
5.2.5 Type-Checker ....................................................... 69
5.3 Automation Part ......................................................... 71
5.3.1 Information Process ............................................... 71
5.3.2 Action Process ....................................................... 73
5.3.2.1 Clone-Action .................................................... 73
5.3.2.2 Rename-Action ................................................ 75
5.3.2.3 Event-Action ..................................................... 76
5.3.2.4 Update-Action .................................................. 76
5.3.3 Device Process ....................................................... 78
5.4 Plugin Part .............................................................. 80
5.4.1 Skeleton Program ................................................... 81
5.4.1.1 Variable-Block ................................................ 81
5.4.1.2 Statistics-Block ............................................... 82
5.4.1.3 Rules-Block ..................................................... 83
5.4.2 Plugin Process ....................................................... 85
5.4.2.1 Updating Variables & Validating Rules .................... 85
5.4.2.2 Generating Rule-Codes ....................................... 87
5.4.2.3 Running Daemon-Plugin as a Linux Process ............. 89
5.5 Summary of Buna Design .............................................. 90

6 "buna": Implementation .................................................. 92
6.1 Configuration File ..................................................... 92
6.2 Rules File ............................................................. 93
6.3 Option-Manager ....................................................... 93
6.4 Name-Generator and Filter .......................................... 96
6.5 Type-Checker .......................................................... 98
6.6 Automation Part ......................................................... 98
6.6.1 Information Process ............................................... 98
6.6.2 Action Process ....................................................... 99
6.6.2.1 Clone-Action .................................................... 99
6.6.2.2 Rename-Action ................................................ 102
List of Figures

2.1 Xen 3.0 Architecture; Dom0 controls any driver request from other VMs, the hypervisor manage and secure physical resources, and VM schedules [24] ......................................................... 17

2.2 KVM Architecture: Each VM has a user mode that used to execute user commands and QEMU-KVM mode that controls I/O requests to access emulated hardware. The KVM-kernel is responsible to create VMs, switch their mode, assign VCPU for VMs [29] . . . . 19

2.3 The left side shows hypervisors without libvirt where as the right side shows hypervisors are working with libvirt software . . . . 20

2.4 Different hypervisors working in same host with the support of libvirt APIs (drivers) ........................................................... 21

2.5 Different hypervisors working in multiple hosts with the support of libvirt APIs (drivers) and daemon(libvirtd) .................. 22

2.6 Architecture of MLN. At the top there are resources grouped into inherited class, at the middle grouped VMs take resources from class and at the bottom all groups and VMs distributed to servers; where one server works as a manager of the others. .................. 25

2.7 Red Hat Enterprise Virtualization Management architecture that shows its communication between users, storages and hypervisors [32] ....... 29

2.8 Home page of RHEV manager [32] .............................. 30

3.1 libvirt application objects exposed to application tools which is used to manage VMs [7] ......................................................... 34

3.2 It shows when an application request an ssh connection to Qemu hypervisor located at a remote host called host1 ..................... 36

3.3 Graphical interface of virt-manager tools. By right-clicking on each domain it is possible to start, stop, migrate, clone and configure the domain [29] .................. 44

4.1 Generating a meaningful name for a VM can be done by concatenating all/some substring existing through a path beginning from the root to each leaf node. ................................................. 49
LIST OF FIGURES

4.2 The figure shows the approach to manage resources between hosts in an efficient way. The doted-square-area shows the skeleton program while it is modified with rules-file to generate the complete daemon-plugin. ................................................................. 53

5.1 Context Diagram of buna. buna interacts with external user directly with command options or indirectly through storage files, and interact with libvirt APIs through commands. ............... 57

5.2 High level structure of buna, input options will be sensed by subprocesses and forwarded for further processes if they are free from error. Further processes are plugin, information, automation and device. ................................. 62

5.3 Flowchart for updating global variables like host-uri, VMs image and snapshot storages path through buna.conf configuration file. ................................. 64

5.4 Flowchart of Name-Generator: generates the full names of VMs from the input given through --vm and feeds the result to Name-Filter. ................................. 66

5.5 Flowchart of Name-Filter: filters the existing VMs from hosts based on the names collected from Name-Generator and then returns them to Option-Manager. ......................................................... 68

5.6 Flowchart for Type-Checker process: It decides which process is going to be triggered to process the users task. .................. 70

5.7 DFD of Information-automation: It collects information about VMs through virsh command and returns the result to users. .................. 71

5.8 Flowchart of Information-Automation: That is used to gather information about VMs based on the values given for --information and --vm options. ......................................................... 72

5.9 DFD of Action-Automation process, it collects all names of VMs and apply the given action on each VMs by using virsh/virt-clone commands. ......................................................... 73

5.10 Flowchart of Cloning-Automation process, it generates a virt-clone command for each VM name and runs on libvirt API. ............... 74

5.11 Flowchart of Rename-Automation: It is used to change the names of virtual machines, which has a grate impact for regrouping VMs. ................................. 75

5.12 Flowchart of Action-Automation: by using the value given through -- automation option, the appropriate virsh action command will be generated for each VM and executed on libvirt API. ............... 77

5.13 DFD of Device-automation process: It automates the virsh command to attach/detach devices on each virtual machine. .................. 78

5.14 Flow chart for Device-Automation: that shows a way to attach/detach devices on virtual machines. ......................................................... 79

5.15 DFD of plugin-process: A daemon-plugin program is generated by combining the skeleton program and rules-file. The program will run as a Linux process that is used to control resources between hosts. 80
5.16 Flowchart of buna skeleton program that has three separated block-codes named variable, statistics, and rule block. 84

5.17 Flowchart of plugin-process: the process takes the names of used hosts, modifications of some variables, and checks the syntax of rules written in the rule file. 86

5.18 Flowchart for generating rules-code (if-condition-action code) for each rule of rules-file. 88

5.19 Context Diagram of buna. This image shows the full structure of buna interaction with users and libvirt APIs, and data flows between internal processes. 91

7.1 Testbed for buna experiment: both hosts are installed with ubuntu-server-10.04 and libvirt. H1 is configured with buna, and also the template VMs to NSA department are hosted here. Similarly H2 is ready for Biology and Chemistry VMs, and H3 is for Architecture department VMs. 119
List of Tables

2.1 libvirt supported hypervisors. . . . . . . . . . . . . . . . . . . . . 23
3.1 Connecting to local and remote hypervisors. . . . . . . . . . . . . 32
5.1 Grouping buna option values. . . . . . . . . . . . . . . . . . . . . 63
5.2 Variables, that are used to store statistics results, are generated by concatenating elements of the three sets. . . . . . . . . . . . . 83
7.1 Sample plan for the university scenario. . . . . . . . . . . . . . . 116
Acknowledgements

This thesis work wouldn’t have succeeded without the help, support and encouragement of my supervisor, teachers, & friends.

I am so thankful to my advisers Amir M. Ahmed and Hårek Haugerud for their ideas, helpful discussions and encouragements. I also would like to use this opportunity to highly appreciate & extend my gratitude to all NSA department instructors & administration staff for their kind support, help & services I have been getting throughout my stay in the school.

I am also grateful to my friends who have been with me devoting their precious time in giving insightful ideas and dedicately editing this thesis paper. Especially, Daniel H., Ephrem T., Wondimagegne K., Addisu Y., Takele A., Assefa M. & Solomon L. deserve my special appreciation & gratitude for their special support starting from the beginning to succeed my study.
LIST OF TABLES

Keywords: buna, automation, plugin, libvirt, Virtualization, VM management.

Abbreviations:

- **VM**: Virtual Machine
- **MLN**: Managing Large Area Network
- **RHEV**: Red Hat Enterprise Virtualization.
- **libvirt API**: libvirt Application Programming Interface
- **DFD**: Data Flow Diagram

Glossary

- **buna**: is a VM and Host administration and management tool under libvirt environment, which is newly created and developed in this project.
- **VM Management**: includes creating, starting/stopping, deleting, cloning, updating/upgrading, attaching/detaching, load-balancing, and getting necessary info about the status of a VM.
- **Host**: is a physical machine where VMs are deployed.
- **VM**: is a software based machine managed by hypervisors.
- **Hypervisor**: is a software (like Xen, KVM, VMware) which allows multiple OSs (called VMs) to run on a single host and shares the host resources (CPU, RAM, DISK, NIC, I/O, etc) for VMs.
- **libvirt API**: a tool-kit that creates a common application interface for external application tools or users to get access to most hypervisors.
- **Property**: is a common name for both host resources (like CPU, Memory, Ethernet) and the status of virtual machines (like running, idle and down).
- This thesis uses the word user & administrator interchangeably.
Chapter 1

Introduction

1.1 Motivation

Virtualization is one of the current indispensable technologies in the computer industry. It is a method of creating fully functional software-based VMs (commonly referred to as guest OSs) that can give equivalent service as a physical computer [19, 21]. There are different types of platforms (like Xen, KVM and VMware) called hypervisors. Each hypervisor has a special benefit and has its own tools that give a common VM management service (sec. 2.1.1). Managing includes creating, starting, stopping, migrating and updating VMs, and controlling resource usage from hosts. When the number of VMs increases, managing individual VMs becomes a problem for administrators. For this, administrators need automating tools that can do routine tasks with minimum human intervention.

There are known and effective tools working in VM management automation, such as MLN, RHEV, VMware vCenter and VMware vSphere (sec. 2.3). These tools minimize routine and time-taking tasks, but all of them have the following drawbacks:

1. They are limited to specific hypervisors, and some of them are commercial (sec. 2.3).

2. They lack simplicity and flexibility to be applied in some organizations like universities, colleges and training centers, with dynamically changing VMs. For example, Universities accept and graduate thousands of students each year, students are adjusted to departments, in each department there are students at different levels. To use these tools, every time, thousands of VM names must be adjusted and added into groups. It is clear how much the overload will be for administrators (sec. 3.10).

3. They don’t have features or not flexible enough to be used for managing resources between hosts (like CPU and Memory) and controlling statuses of virtual machines (like running, idle and down).
1.2. PROBLEM STATEMENT

On the other hand, *libvirt* is a free software that creates a *Common Application Programming Interfaces (APIs)* for most virtualization platforms. That means, application tools developed by using libvirt APIs will work on all libvirt supporting hypervisors (KVM/QEMU, Xen, LXC, OpenVZ, User Model Linux, Virtual Box, VMware, UML)[7, 20, 16]. This eases the burden of understanding and operating different hypervisors’ managing tools. There are known VM managing tools that are working on libvirt such as *virsh* [10], *virt-clone* [11], *virt-install* [12], *Virt-manager* [12] and *guest-fish*. All these managing tools give complete administration on VMs, but they lack an *automated management system* to ease the routine tasks of system administrators.

Considering the fact that libvirt supports the majority of current hypervisors, implementing an automated management system using libvirt environment will be a considerable enhancement in solving the above drawbacks.

1.2 Problem Statement

Effective management of virtual machines is a crucial part of system administration. As the number of virtual machines in an organization increases, management of the virtual machines gets more and more complex. This will add too much load on the day to day tasks of system administrators; which in turn affects the well being of the IT system in general. To deal with such problems, automation of the manual management process is an effective solution. There are some automated virtual machine management tools that are already in use. Basically these tools are limited to specific hypervisors; don’t manage VMs with respect to the resources usage on hosts, and they lack consideration of the needs of organizations with a dynamically changing virtual machine environment.

1.3 Scope and Limitations

This thesis works on providing a solution for problems stated above. It focuses on devising a *simple*, *flexible* and *powerful* virtual machine and host automation management system, especially for dynamically changing environments. As a proof of concept a tool named *buna* is developed that works on libvirt APIs.

The functionalities of *buna* are limited to the functionalities provided by libvirt managers: *virsh*, *virt-clone* and *virt-top*. So, fine-grained control of VMs, like setting IP address, installing packages and changing user names are not included. It’s possible to enable usage of additional libvirt mangers with *buna* to control VMs at a fine-grained level, but this thesis is limited to 17 weeks work and that is not
1.4 Approach

This thesis is an investigative research work given the fact that it suggests solutions based on well defined mathematical set theory, logic and descriptive statistics. And the solutions are applied on the development of the prototype tool buna. There is a basic approach difference in the naming of VMs compared to existing VM management tools, which makes buna simple-to-use, flexible and powerful. The tool is developed in the Perl programming language as it is powerful in string manipulation and file handling.

buna has two major parts: automation and plugin. The automation part deals with automating the VM management tasks. The plugin part enables controlling resources of hosts and statuses of virtual machines by using the automation part.

1.4.1 Automation Part

Using the existing VM management and automations tools, each VM is managed either one by one or VMs name must be listed down into files/groups; which increases the management complexity and is also time consuming.

But by giving a logical and meaningful name for VMs, it's possible to create mathematical relationships between VMs. This can be done by adding common substring on the name of VMs. When there is a need to access these VMs, one can filter them from the whole through their relationship (common substring).

To implement this idea, the Cartesian product from mathematics set-theory is applied together with the text manipulating capabilities of the Perl programming language. This approach makes the automation system to be done with a simple command, and avoids the need to refer group files or complex configuration files (sec. 4.1).

1.4.2 Plugin Part

The plugin part controls resources between hosts and statuses of VMs. Normally, every host has resources like CPU, Memory and Network. Virtual machines residing on hosts will be in one of the states like: active, inactive, running, blocked and paused. This thesis refers both the resources of hosts and the status of virtual machines as properties.
1.5. RESEARCH GOALS

By periodically collecting data about these properties from hosts and finding descriptive statistics from the data (like minimum, average and mode), it is possible to take automatic actions based on the fulfillment of some conditions; where conditions are numeric comparative expressions written with respect to the collected statistics.

To implement this idea descriptive statistics and logic will be applied, where the mathematics logic makes writing rules user-friendly.

1.5 Research Goals

The goal of this thesis work is to facilitate the VM management and minimize the burden of system administrators by applying a novel mathematical approach and design and develop an Automated Virtual Machines and Host Management System named buna.
1.6 Thesis Structure

The structure of the thesis is as follows:

- **Chapter-1**: Introduced readers about virtualization, the existing VM managements and libvirt. Problems on existing VM management tools were stated and a new approach to deal with such problems was discussed. The scope, limitations and goals of the thesis were also discussed here.

- **Chapter-2**: Provides background information about the existing virtualization platforms, their VM management systems and how to use them with libvirt managing tools. The state-of-the-art of existing automation tools with their prons and cons are explained here.

- **Chapter-3**: Detailed explanation of libvirt tool kit is shown, including but not limited to its installation, functionalities, object(VM, device, network, ...) representation format and existing libvirt application VM managers(virsh, virt-clone, virt-top, guestfish, ...) with their drawbacks.

- **Chapter-4**: Explains the mathematical approach suggested by this thesis to deal with the identified problems in virtual machine and host management.

- **Chapter-5**: The design of the prototype tool (buna) is set in detail with the support of DFD and Flowcharts.

- **Chapter-6**: Discuss the implementation of buna. Basic parts of buna Perl codes are explained; the full program is attached in the appendix.

- **Chapter-7**: Deals with buna testing based on a university scenario. The functionalities of buna are tested & resulted both in actual & simulated form.

- **Chapter-8**: Discusses about alternative approaches, achieved goals, and some limitations of buna.

- **Chapter-9**: Contains the conclusion statements of this thesis.

- **Chapter-10**: Contains suggestions for further work in the area of libvirt and virtualization. From all, the one stated at the last position can be taken as a master thesis.

- **Appendix-A**: Contains the full program of buna with its configuration file (buna.conf) and a default rules file. Some detail information related to this thesis are also included here.
Chapter 2

Background and Literature Review

This chapter focuses on general explanation of virtualization (Xen and KVM hypervisors), libvirt tool kit, and presents investigation on existing automated administration and management tools.

2.1 Virtualization

Virtualization which is started in the 1960s by IBM [27, 28], is one of the current and very common applicable technologies in the computer industry. It is a method of creating a fully functional software based computers called VMs, where this VMs will give an equivalent service as an ordinary computer. Hence each VM operating system will have its own specific virtual resources like CPU, memory, storage, Ethernet, virtual I/O [17, 26]. All resource allocation and VMs management tasks will be the responsibility of virtual machine manager software called hypervisor. There are number of Open source and commercial hypervisor platforms, the common ones are: KVM, Xen, VMwere, OpenVZ, Oracle VM. Where KVM, OpenVZ and Xen are open source [19].

To get a clear understanding of virtualization it is better to see the two main open source hypervisors that have a clear architectural differences: Xen and KVM. This two hypervisors can represent the current existing approaches of virtualization technologies.

2.1.1 Xen and KVM Overview

Architectural, compatibility and performance view of Xen and KVM will be explained in this section.
2.1. VIRTUALIZATION

2.1.1.1 Architecture

Xen

Xen is an open source platform developed at Cambridge University, UK. It supports para virtualization; to work as full virtualization, Xen needs CPUs that support virtualization [31, 18]. In Xen platform, all virtual machines, except Domain0, don’t have direct access to physical resources; they will be set as a level 3 user. Domain0 is responsible to manage drivers and I/O requests to other virtual machines [30]. Any virtual guest operating system will send request to domain0, when they need to access a device driver (table 2.1). The Xen Hypervisor will be set as a level 1 user, and it manages both CPU, memory, powers and scheduling of virtual machines[26]. Xen has two scheduling approaches: Simple Earliest Deadline first Scheduler and Credit Scheduler [31].

![Xen 3.0 Architecture](image)

Figure 2.1: Xen 3.0 Architecture; Dom0 controls any driver request from other VMs, the hypervisor manage and secure physical resources, and VM schedules [24]
2.1. VIRTUALIZATION

**KVM**

KVM (Kernel Virtual Machine) is one of the latest open source full virtualization platforms invented and sponsored by Qumranet [28, 9]. Its development focuses on the new architecture hardware devices like Intel VTX and AMD-V CPUs that support virtualization at hardware level [33, 26, 23]. KVM gets the advantage of being the late comer to see the approach, benefit and drawbacks of Older open source platform like Xen [23]. In addition to Linux, KVM is broadly supported by RedHat, AMD, IBM, Intel, Nobel and others. At this time KVM has vast number of customers spanning from small to large companies [33, 26].

In KVM, all virtual machines are considered as Linux processes like MySQL and PHP. Which shows VMs can be killed like any process. Each VM has its own virtual hardware resources; in fact KVM sets virtual CPUs a process to be used by VMs [22, 25]. As can be seen on figure 2.2, KVM has a modified QEMU for each VM that controls I/O services for invoking processes given by KVM kernel module [36, 29]. KVM adds a third execution mode called guest mode in addition to the kernel and user modes. The guest mode has its own kernel and user which lets it to work as a normal machine [35, 36]. When the guest requests to get execution time, the kernel makes resources ready. When the guest finishes execution, the kernel will save the guest and shift back to host mode [23].

This architectural deference between Xen and KVM exists because Xen was created when most hardware CPU didn’t implement virtualization intelligence, while KVM is invented on new CPU generations that implement virtualization [22]. This lets KVM to concentrate on how to manage VMs rather than focusing on hardware emulation.

**2.1.1.2 Compatibility**

**Xen**

Xen supports x86-64 architectures, means it can work both in old and new CPU architectures [33, 26, 31]. Linux kernel by itself didn’t include Xen at all; modifying and shipping Linux with Xen is the responsibility of Xen distributors. Xen host operating system must be Linux and Unix groups, but hosts could be Linux or Windows Operating System [36, 27, 22].

**KVM**

KVM is a based on x86 hardware platform with latest CPUs that support virtualization: Intel VT, AMD-V, IBM, Nobel [33, 26, 28]. KVM supports Linux, Windows, Solaris, and BSD based virtual machines but the host operating system must be Linux with versions from kernel 2.6.2.3 [22, 28].
2.1. VIRTUALIZATION

Figure 2.2: KVM Architecture: Each VM has a user mode that used to execute user commands and QEMU-KVM mode that controls I/O requests to access emulated hardware. The KVM-kernel is responsible to create VMs, switch their mode, assign VCPU for VMs [29]

2.1.1.3 Performance

Xen
Most researches show that Xen’s performance is good. During performance comparison between OpenVZ, VMWere and Xen performance, Xen performs better in CPU usage, memory management, and fork services but a little bit slower in bandwidth consumption and Gziping [33, 18]. Scheduling performance of Xen shows good result with high throughput in non-real time applications but there is evaluating difficulty in low throughput real time applications [31]. Xen has guaranteed to be the best for para virtualization bases [33].

KVM
Lots of researches showed that overall KVM performs better in I/O services when compared to Xen and VMWere [33], great throughput performance was shown on KVM when compared with Xen [27], KVM performance is better in full virtualization bases [33].
2.2 Libvirt Virtualization Modes

Libvirt is a group of C programs that creates a common application interface for all hypervisors. When libvirt is installed with a hypervisor, the hypervisor will be set at the back end and libvirt will create an application interface called API for managing VMs. So any user or application tools will use the API for managing VMs [6, 16].

2.2.1 Single Hypervisor

When libvirt is installed together with single hypervisor, it creates a common interface for external users to communicate with the hypervisor. Using libvirt for this case holds no value, rather it overloads the system, because the hypervisor manager itself can do everything needed. Figure 2.3 shows how the hypervisor, libvirt and management application tools are working on a single host.

**NB:** *In libvirt Virtual machines are called Domains.*

![Figure 2.3: The left side shows hypervisors without libvirt where as the right side shows hypervisors are working with libvirt software](image)

2.2.2 Multi Hypervisors

Libvirt can manage a number of hypervisors that run on same host; in this case libvirt has a collection of API drivers that allows external applications to get access to all hypervisors, but accesses is possible to only one hypervisor at a time (see figure 2.4).
2.2. LIBVIRT VIRTUALIZATION MODES

Figure 2.4: Different hypervisors working in same host with the support of libvirt APIs (drivers)

2.2.3 Multi Host Mode

In addition to different hypervisors, libvirt supports management of distributed hypervisors on a number of hosts. That means the management application tools installed in a host can manage VMs on other hosts. For this libvirt has a daemon called \textit{libvirtd}, which runs on a host that doesn’t have application management tools. When application management tools request a domain on other hosts a request will be sent to that host with the daemon and the daemon will process the task and give the response back to the application manager host [6]. See the figure 2.5

2.2.4 When to use Libvirt?

The main target of libvirt is to use the same management tools for different hypervisors. Hence it is not advisable to use libvirt if the system has only one host and one hypervisor, the hypervisor itself can manage VMs. But it is advisable to use libvirt on the area that different hypervisors and number of hosts exist; this will help administrators to use a single tool for all. (Table 2.2.4 shows list of hypervisors supported by libvirt.)
2.3 Virtual Machine Management Automation Tools

In short virtual machine management means controlling VMs. The following are functionalities in virtualization management system.

- **Basic features**: The very basic features of virtualization management are able to create, delete, start, stop, halt, store, ... virtual machines.

- **Migration**: A capability of moving CPU and Memory snapshots of running VMs from one host to another. Live migration is an advanced feature that enables migration of VMs without interrupting their execution.

- **Power Saving**: This is the capability of collecting running virtual machines to as few hosts as possible to accommodate the VMs so that more hosts can be idle.

- **Maintaining**: It is the capability of fixing VMs while they are live.

- **Image Template**: The capability of using template images to create new VMs. This will save configuration time, system usage and power. This feature is important to create a number of VMs at ones.
### 2.3. VIRTUAL MACHINE MANAGEMENT AUTOMATION TOOLS

<table>
<thead>
<tr>
<th>Hypervisor Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Qemu</td>
<td>supports latest architectures that have CPU virtualization capabilities</td>
</tr>
<tr>
<td>KVM</td>
<td>Kernel base virtualization.</td>
</tr>
<tr>
<td>Linux Containers</td>
<td>Linux light weight virtual system</td>
</tr>
<tr>
<td>OpenVZ</td>
<td>Linux operating system level virtualization</td>
</tr>
<tr>
<td>VirtualBox</td>
<td>Hypervisor support X86</td>
</tr>
<tr>
<td>UML</td>
<td>User Mode Linux</td>
</tr>
<tr>
<td>Storage</td>
<td>storage pooling for local or network disks and iSCSI</td>
</tr>
</tbody>
</table>

Table 2.1: libvirt supported hypervisors.

- **Security**: Protect the VMs from unauthorized access.
- **Interface**: Good user interface for external users (can be shell command, GUI or web interface).
- **Centralization**: Capability of managing distributed VMs on different hosts from a single host.
- **Availability**: Keeping the VMs running after a failure by using snapshots.
- **Scalability**: Capability of managing large virtual machines and their infrastructure.
- **Automation**: Capability of performing the above listed functionalities automatically.

Current virtual machine management tools fulfill most of the above features, but still there are only few tools (like RHEV manager) that are capable of automation, and power management. Even though most management tools didn’t have automation, Command based management tools are flexible for shifting them to automatic management system, but GUI and web based ones are expected to be changed from the ground. MLN is the one that uses this chance to have automated management in Xen, UML, and EC2 platforms.

### 2.3.1 MLN Manager (Linux open source)

#### 2.3.1.1 Functions

*Manage Large Area Network (MLN)* is an open source virtual machine management tool, developed in 2004 using Perl programming; it is designed to automate
large virtual networks managements in Xen, UML, and Esxi platforms. Using MLN it is easy to create, start, stop groups of VMs and to move VMs from one network to another within a matter of minutes \([2, 34]\).

2.3.1.2 Architecture

In MLN all Virtual machines are categorized into groups called projects; each project has specific file and name. Then MLN manages virtual machines based on the information of each group.

In each project file the project name and names of virtual machines must be stated. Each virtual machine will be assigned with resources like CPU, Memory, Storage, Network address, user name, image file (installed OS going to be used as template). To solve the burden of writing all resources for each VM, common resources that will be given for a number of VM’s can be stated in an inheritable class and included in each hostclasses. MLN class follows Object Oriented programming approach \([15]\). See figure (2.6).

This approach of MLN lets it to create, start, stop and migrate VMs in project level. The boundary of MLN management is extended from single machine to groups of VMs in a project.

2.3.1.3 How to use MLN?

To use MLN, first a project file must be created based on MLN syntax. This file contains the whole structure of VMs and their network and groups. Then using MLN command the file will be changed into real VMs and network (referred as building a project). After the real network is deployed in to the system, it is possible to turn on/off and migrate VM groups in between different servers \([15, 34]\).

Here below are steps of using MLN: -

- Plan the VMs, Servers and the network which are going to be created
- Write that into a file called project file, and save it with .mln extension.
- Build the planned project in to real VM and network using mln build command
- Then manage the network by starting, stopping, and migrating them.

2.3.1.3.1 Creating Project Files: Here is an example on how to create a project called example which has one VM called VM1. When this project is built, VM1 OS will be taken from the template, 128M memory, 2G storage, 1 CPU will
2.3. VIRTUAL MACHINE MANAGEMENT AUTOMATION TOOLS

Figure 2.6: Architecture of MLN. At the top there are resources grouped into inherited class, at the middle grouped VMs take resources from class and at the bottom all groups and VMs distributed to servers; where one server works as a manager of the others.

be assigned and its network configuration will be done based on the information.

---

Sample MLN project file

```
global {
    project example
}
host VM1 {
    xen
}
```
2.3. VIRTUAL MACHINE MANAGEMENT AUTOMATION TOOLS

MLN has its own block keywords named **global, host, switch, network and users**, which are used for representing objects. Other key words like **memory, cpu and template** can be set inside block key words (appendix A). This keywords have key and value attributes. In short block key words are as like objects and others are as attributes of an object.

The structure of the file is similar to object oriented programming; there is inheritance from one to another block. A block is defined with the keywords and curly braces, where as attributes are defined inside the curl bracket with key and value combination. There are some attributes that do not have value like **xen, uml** which shows that this project will be built in either Xen hyperviosr or UML. The full structure of project file is shown in appendix ’A’.

**2.3.1.3.2 Building VM’s & Networks:** After creating the project file the next step is building the network based on the project file. Run the **mln** command using **build** option as follows.

```
mln build -f localnet.mln
```

Before building the project, this command checks whether the project file is written in MLN syntax and then it checks the amount of memory and storage to deploy the project against the available hardware resources.

When MLN builds projects, it will copy the template file to each individual VM and do the configuration change based on the VM information in the project and save it with the the VMs name in the projects directory on the destination server; in addition the configuration file and **start** and **stop** bash scripts of each VM will be created.

```
Ivm
memory 128M
template ubuntu-server.ext3
size 2GB
nameserver 10.0.0.15
network eth0 {
    address 10.0.0.2
    netmask 255.255.255.0
    gateway 10.0.0.1
}
users {
    somebody te47-dikkjk
}
```
2.3. VIRTUAL MACHINE MANAGEMENT AUTOMATION TOOLS

2.3.1.3.3 Managing Projects: After building projects using the above steps, managing the existing projects will be easy for MLN. Managing projects will be done by using the following MLN commands.

<table>
<thead>
<tr>
<th>Managing projects using MLN command</th>
</tr>
</thead>
<tbody>
<tr>
<td>To Start a project: mln start -p project-name-here</td>
</tr>
<tr>
<td>To Stop a project: mln stop -p project-name-here</td>
</tr>
<tr>
<td>To get info about a project: mln status -p project-name-here</td>
</tr>
<tr>
<td>NB: to address a single VM add the following <code>-H host_name</code></td>
</tr>
</tbody>
</table>

2.3.1.4 Adding Plugin

MLN has a framework that allows researchers to add any type of plugin that can work for automating hypervisors. For the time being MLN supports Xen, UML (User Mode Linux), VMware Server platform and Amazon EC2 systems. So it is possible to add plugin for non-supported hypervisors like KVM.

To use MLN, administrators have to know a number of keywords and the basic structure of writing a project.

2.3.2 RHEV Manager (Red Hat commercial)

2.3.2.1 Functions

Red Hat Enterprise Virtualization (RHEV) manager is commercial web interface based centralized management system that supports RHEV-H and Red Hat Enterprise Linux 5 or latest with KVM virtualization platform. RHEV manager has the capability of managing hosts, guests and storage media. In addition to the common features of virtual management, RHEV adds advanced feature to manage VLANs [32].

Using RHEV manager it is possible to:

- group VM’s called clustering, which makes the management like changing Memory size, migrating, installing a software, ... easier.
- filter VM’s with their host, network, CPU usage, operating system.
- live migration from host to host when it is needed.
- manage all VMs residing in different hosts at a central place (figure 2.7)
- manage storages like NFS, iSCSI, FCP.
2.3. VIRTUAL MACHINE MANAGEMENT AUTOMATION TOOLS

- adjust VM’s into a power save mode.
- create templates that can be used for quick duplication of identical VMs.

**NB:** But RHEV doesn’t have the capability of creating VM’s in mass and in an automatic manner; VMs have to be created one by one. But managing them, after creation is automated. However, MLN has this automation capability.

### 2.3.2.2 Architecture

RHEV manager runs only on AMD (AMD-v and AMd64) and intel(intel IV and 64) CPU extensions. To run the client side RHEV on windows 2008, it needs SQL server, IIS enabled, Putty and WinScp.

All VMs, storages together with their hypervisor will be set at the back end; RHEV manager will be between hypervisors and external users. The manager uses LDAP, Active Directory and Database to control and manage the communication between the hypervisor and end user. RHEV manager has different types of interfaces that can be categorized in to two:

- **Web server** used by administrators and users to get access to RHEV server and RHEV client environments from web browsers.
- **Rest and Sops** systems that are used for administrators to get access through power Shell, Python and WPF.

RHEV manager supports the following operating systems to run on virtual machines:

- Red Hat Enterprise Linux 3 and newer with 32 bit and 64 bit
- Windows XP SP3 and newer for 32 bit only
- Windows server 2003 SP2 and newer
- Windows server 2008 and Windows 7

### 2.3.2.3 How to Use RHEV?

RHEV has two types of web interface ports named RHEV Admin and RHEV User. The RHEV Admin console is used to manage VMs and storages; whereas RHEV User allows specific users to access their own VMs.
2.3. VIRTUAL MACHINE MANAGEMENT AUTOMATION TOOLS

Figure 2.7: Red Hat Enterprise Virtualization Management architecture that shows its communication between users, storages and hypervisors [32].

2.3.2.3.1 RHEV for Administrators: This interface shows a list of existing VMs and clickable buttons to see or add data centers, clusters, hosts, storages, VM’s, Pools, templates and users. The central part of the page is used to show selected VM’s name, node, storage and status. At the top there is a search bar that used to search existing VM’s, storages and hosts with different characteristics. For example "VM’s: WindowsXP and apps=RHEV-USB" is used to search virtual machines running WindowsXP and have RHEV device installed (see figure 2.8)

The ‘new’ button is used to add new user or storage pools. It has options to select template name, to insert VM name and description, to choose the cluster/group, default host, storage, memory, number of CPU, Operating system and check/uncheck option about availability level. The highly available option lets the VM to be stored in different nodes to increase its availability.

2.3.2.3.2 RHEV User Console: This interface allows users to access their own virtual machines. Users can access VMs from browsers by using the server name followed by their ID.
2.3. VIRTUAL MACHINE MANAGEMENT AUTOMATION TOOLS

2.3.3 Other Automation Tools

There are VM automation tools working in the cloud such as *VMware vSphere CLI*, which is a command line that is working for ESX and ESXi host; *VMware vCenter* which automates resource usage and configuration; *Opalis intros Virtualization* is used to automate fault managements, backup and recovery operations [8].

**NB:** Because libvirt is the core part for the development of this paper it needs detailed explanation. The next chapter, "Libvirt Software", comprises an extended discussion of Libvirt.
Chapter 3

Libvirt Software (Tool-kit)

3.1 Overview

Libvirt is free software, which consists of a number of programs written in C, which creates a common Application Programming Interface (API) for most virtualization platforms (KVM/QEMU, Xen, LXC, OpenVZ, User Mode Linux, Virtual Box, VMware, Microsoft Hyper-V). The main goal of libvirt is to create a common environment called API to manage VMs in all platforms and reduce the burden of creating application tools for each virtualization platform. It is free and available under GNU Lesser General Public License. Basically libvirt have been working on Linux OS; for windows the trial version is released this year [7]. Libvirt was originally designed for Xen but now it pools most of virtualization platforms [6, 20]

3.2 Installation

It is possible to download libvirt form http://www.libvirt.org FTP and Http Servers. It is also possible to get from git repository; here a hourly snapshots are taken, so there is no stability grant.

```
  git clone git://libvirt.org/libvirt.git
```

From the repository, it’s possible to download by using `apt` and `yum` commands. From the tar-ball it has to be untard by using `gzip` command and use `make and install` tools.
3.3 CONNECTING TO LIBVIRT API

3.3 Connecting to Libvirt API

After installation has completed it is possible to connect to any hypervisor (Qemu/KVM, Xen) through libvirt API. There are different application tools (like virt-install, virsh, virt-clone) that uses libvirt APIs for VM management. To connect to libvirt APIs using command line managers looks like

\[
< \text{command to be used} > < \text{hypervisor uri} >
\]

; here the first part must be the application tool that is going to be used, the second part is the URI of the machine where the hypervisor is located.

The syntax of URI looks like as shown on the table 3.3.

<table>
<thead>
<tr>
<th>Local URI</th>
<th>Remote URI</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>xen:///</td>
<td>remote:/host-xen/</td>
<td>creates connection to Xen hypervisor from local and remote machine called host-xen.</td>
</tr>
<tr>
<td>NULL</td>
<td>remote:/host-xen/</td>
<td>creates connection with the default hypervisor running on local and host-xen machine.</td>
</tr>
<tr>
<td>xen:///</td>
<td>Xen+ssh://host-xen/</td>
<td>use ssh for creating the connection.</td>
</tr>
</tbody>
</table>

Table 3.1: Connecting to local and remote hypervisors.

Here below are some examples that uses virt-manager tools to connect to hypervisors.

using virsh connect to default host which was set to environment variable VIRSH_DEFAULT_CONNECT_URI

virsh -c test:///default list

# connecting to remote host Xen hypervisor through ssh
virsh -c xen+ssh://remote-machine-name or ip/

virt-manager -c test:///default

3.4 Firewall and Network Filtering:

Libvirt supports firewall and network filtering capabilities at a high level. Still there are two ways of network and filter options, called virtual network driver and network filter driver.
3.5. LIMITATION OF APIs

- **Virtual network driver:** creates a software based virtual bridge which is independent on any physical NICs. All domains virtual network port will going to connect to this virtual bridge and any filter rule will be attached to each domain network port. This virtual bridge allows all outgoing requests and internal communications between virtual machines, and blocks a request coming from outside; but NATing is possible.

- **Network filter driver:** the driver is fully configured capability that could accept any IP table rules (IP4, IP6). The rules have to be written the XML format. The example below drops all outgoing requests with priority of ’500’.

```xml
<filter name='no-spamming' chain='XXXX'>
  <uuid>d217f2d7-5a04-0e01-8b98-ec2743436b74</uuid>

  <rule ...>
    <rule action='drop' direction='out' priority='500'>
    ....
  </rule>
</filter>
```

There is also a new third idea called 'QEMU driver MAC filter'. This focuses on filtering MAC address to prevent spoofing attacks

### 3.5 Limitation of APIs

APIs are limited to access one Node at a time and have the right to access resources that are used for managing domains like, Networking, firewall, storage and CPU. All APIs can access the node remotely. APIs are not allowed to apply multi node managements policies like load balancing [7, 16].

### 3.6 Security and Stability

Libvirt has Libvirt-CIM that provides a virtualization platform for Linux operating system which supports Common Information Model Object Manager. It supports security techniques like TLS encryption and certifications, remote management authentications using Kerberos and SASL is also available. Libvirt has strong base that will keep working all application tools without modification even if there exists modification on libvirt system [7, 6].
3.7 Libvirt Architecture

Libvirt loads all hypervisors on top of it without disturbing them; then it will create an interface through API to external users. Libvirt supports management of distributed hypervisors on multiple nodes [20]. That means the management application tools installed in a node can manage all other nodes. For this libvirt has a daemon called libvirtd, which is going to be run on nodes that do not have application management tools. When application management tools request a domain on another node, a request will be sent to that domain through network, the daemon will process the task and give response to the application manager node[6, 20]. See the figure 2.5

3.7.1 Application Programming Interface (API)

Libvirt handles all resources existing in the host domain as object form and expose the objects those are necessary to manage virtual machines to the requesting application, then application tools can use this objects to manage virtualization on any platform. Libvirt expose objects in the XMl format (see Fig. 3.1). An application is limited to get connection to only one domain at a time on a hypervisor [20]. Here below is a list of objects used to mange virtual machines.

![Diagram of libvirt objects](image)

**Figure 3.1: libvirt application objects exposed to application tools which is used to manage VMs [7]**

From the figure above there are four libvirt objects used for virtualization manage-
3.7. LIBVIRT ARCHITECTURE

- **VirConnectPtr**: it is the main interface to access other objects. All tools have to request for connection through this object and can access equivalent hypervisor drivers.

- **VirDomainPtr**: represent a domain in the hypervisor. It identifies each domain by their *name* (that is unique in the host), *ID* (which is a dynamic ID given to running VMs) or *UUID* which is a permanent identifier through the all nodes.

- **VirNetworkPtr**: represents a network existing in the hypervisor.

- **VirStoragePoolPtr**: represents the storage pool(disk) that all VMs are using.

- **VirStorageVolPtr**: represent a storage volume of a VM

- **VirHostPtr**: presents the physical host hardware resources.

### 3.7.2 Hypervisors

Currently libvirt has drivers for Xen, Qemu for Qemu/kvm, UML, OpenVZ, LXC hypervisors. When an application requests to connect to a specific hypervisor, libvirt APIs searches the right driver from the kit. Then the selected driver starts communicating with the back end hypervisor and gives responses back to the application [6, 20, 16].

### Hosts and Domains

Libvirt has a host object called **VirNodeDevPtr**, which gives information about all hardware resources of existing hosts (like Disk, USB, CD, BIOS, CPU, NIC,Sound and VGA cards). It also has a domain object called **VirDomainPtr**, which gives resource usage summary about each virtual machine.

### 3.7.3 Network

Networking provides a communication media that allow domains to be grouped into subnets and also have external communication. By default libvirt create a virtual network called *default*, any new domain will going to connect to this network unless it is configured to have other networks. This default network is fully software based (does not use physical Ethernets), it uses NATing to get external communication, but domains could not be accessed from outside if worked through this network.
3.7. LIBVIRT ARCHITECTURE

There are two types of virtual networks called permanent and dynamic. In permanent virtual network the network structure (configuration and XML files) are stored in the host permanently. But in the dynamic virtual network, the structure of the network alive only if there are running hosts on the network, otherwise it will be destroyed from the host. The function of network object is called VirNetworkPtr.

To create additional networks existing physical Ethernet devices must be assigned to virtual bridges and then a new network can be created by using this bridge. Once the Ethernet card is assigned to a bridge, it will not be used for other services.

3.7.4 Storage

The storage APs management has two basic concepts called pool and volume. The storage pool is used to manage the whole storage types that could be subdivided in to storage volumes. Storage types can be local disk, NFS, iSCSI, logical volume. Storage volume is some part of defined storage pools, which is going to be given for a number of domains (it is similar to a disk partition) [7, 6, 20]. The represen-


3.8 Libvirt XML Object Definitions

Libvirt uses XML format to represent every type of objects like domains, storages, bridge and network. This section explains about the XML presentation of basic objects.

3.8.1 Domain

Domain is a virtual machine that is created on a hypervisor which is going to be controlled by libvirt applications.

3.8.1.1 General Definitions

The top level general definition of a domain contains:

- **Hypervisor**: Assigning the hypervisor name of the domain, where this domain is hosted (can be xen, kvm, qemu, lxc and kqemu). The example indicates the host is managed by Xen,

- **Host name**: An alpha numeric type that holds the name of the domain, name can include underscore and hyphen characters. This name is unique with in a node. The example below shows the name is VM1 [20]

- **UUID**: a 32 byte global unique identifier, it could be given by manually or by system random generator.

*libvirt uses either the Name or UUID to call VMs for management*

```
<domain type='xen' id='3'>
  <name>VM1</name>
  <uuid>hj7a34531d59iuy32516782e98ab3k9u</uuid>
  <description>template domain</description>
  ...
```

3.8.1.2 CPU

Vcpu definition represents the amount of virtual cpu assigned to a domain. In the example below the domain is assigned one cpu, but if it needs more that one it could take upto two cpus. Here the attribute cpuset contains the list of available physical cpus that could be used to print the virtual cpus.
3.8. LIBVIRT XML OBJECT DEFINITIONS

3.8.1.3 Memory

Assigns the maximum memory and current memory that the VM will get when it boots (measure is in kilobytes).

```
<domain>
  ...
  <memory>524288</memory>
  <currentMemory>524288</currentMemory>
  ...
</domain>
```

3.8.1.4 Life Cycle Controls

Life cycle definitions locate the function going to be run when the domain orders an action. There are three attributes called on_poweroff, on_reboot, on_crash. And also there are numbers of values for them like destroy, restart, preserve, rename_restart. The first line of the example below says when the domain requests power off, the domain will be destroyed, just like unplugging the power cable.

```
<domain>
  ...
  <on_poweroff>destroy</on_poweroff>
  <on_reboot>restart</on_reboot>
  <on_crash>restart</on_crash>
  ...
</domain>
```

3.8.1.5 Device and File System

The device XML block of a domain is used to assign the devices going to be given to the domain for its use, this is just like an administrator enabling and disabling CD rooms, floppy drives for some users.

An attribute called type holds the data type of the device going to be given to the domain; this could be a file, directory, network, block. By default all this types will be exposed to the domain as a disk type, but there is an option called types
that shows in what way the device will be exposed to the domain. Type values are `floppy`, `disk` and `CDrom`.

The device XML block has an attribute called `source` which takes the full path of the file going to be referred; and the target `attribute` is the disk path where the source file going to be mounted as root when the operating system runs. The example below shows the source file is stored in `/var/lib/libvirt/images/` directory as `vm2.img` where as the target is on an `ide` type disk called `had`.

It is also possible to share common files from the host to number of domains by adding some rules on it like read only, rewritable, ... . This option specified inside `filesystem` block with `source` and `target` attributes.

In addition to this main attributes, there are also other options that the device could take such as adjusting boot sequence, allow encryption and type, make read-only.

```xml
<devices>
  <emulator>/usr/bin/kvm</emulator>
  <disk type='file' device='disk'>
    <driver name='qemu' type='raw'/>
    <source file='/var/lib/libvirt/images/vm2.img'/>
    <target dev='hda' bus='ide'/>
  </disk>
  <disk type='block' device='cdrom'>
    <driver name='qemu' type='raw'/>
    <target dev='hdc' bus='ide'/>
    <readonly/>
  </disk>
  <filesystem type='mount' accessmode='passthrough'>
    <source dir='/export/to/guest'/>
    <target dir='/import/from/host'/>
    <readonly/>
  </filesystem>
  ...
</devices>
```

### 3.8.1.6 Network Interfaces

This line is used to give which network the host has to use to get its virtual `eth0` and `eth1`. But the network must be defined in libvirt before going to be assigned to VMs. The example below shows the default network is assigned to this host.

```xml
<interface type='network'>
  <mac address='00:16:36:3c:e0:0e'/>
  <source network='default'/>
</interface>
```
3.8.1.7 Other Definitions

In addition to the above main configuration parts of a domain XM, there are also a number of attributes that are used to access input devices like mouse, hub devices like USB, Graphic and video displays, sound, and log and security functions.

3.8.2 Network

Using libvirt it is possible to have a number of virtual networks, there is a default network called default, where all domains by default going to join it. The network XML definition contains the network name with unique UUID, bridge name where the network going to be deployed and the IP ranges that will be given to domains. The following network example is called students, which is using virbr1 (defined at eth1); any domain that is going to use this network will be given an IP address within 192.168.122.0/24 network.

```
<network>
  <name>Students</name>
  <uuid>23a3cb5a-95f4-58f5-cc7c-caae3d47140a</uuid>
  <forward mode='nat'/>
  <bridge name='virbr1' stp='on' delay='0' /> 
  <ip address='192.168.122.1' netmask='255.255.255.0'>
    <dhcp>
      <range start='192.168.122.2' end='192.168.122.254' />
    </dhcp>
  </ip>
</network>
```

3.8.3 Storage

There are two object parts for storage, pool and volume, where pool represents the total storage area and volume represents the specific area which is given to some domains [20]. The XML structure contains unique name and UUID of the pool, storage type (iSCSI, file, local disk ...) and its path.

3.9 Libvirt Virtual Machine Managing Tools

There are lots of applications that have been developed that uses libvirt APIs, below are the common applications used to manage virtual machines both in command line, web and graphical user interface technologies. *Virtual Machine Manager* is
3.9. LIBVIRT VIRTUAL MACHINE MANAGING TOOLS

the known one which is used to create, delete and configure virtual machines.

3.9.1 Virsh (command line)

Virsh (virsh <subcommand> [args]) is a command line user interface manager for Libvirt that is used to manage one VM at a time.

Getting Information:  It could show the status of VMs (running, paused, crashed or stopped); it also used to editing XML files with no error and shows the CPU, memory and devices used by each VM [10, 5]

<table>
<thead>
<tr>
<th>Id</th>
<th>Name</th>
<th>State</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>vm1</td>
<td>running</td>
</tr>
<tr>
<td>7</td>
<td>vm4</td>
<td>running</td>
</tr>
<tr>
<td>-</td>
<td>vm11</td>
<td>shut off</td>
</tr>
</tbody>
</table>

root@ubuntukvm1:~# virsh list --all

root@ubuntukvm1:~# virsh list

root@ubuntukvm1:~# virsh dominfo vm4

Taking Actions:  Using virsh command, it is possible to create a new VM, migrate VMs between hosts, and change the state of a VM like start, stop, undefine, pause and resume.

command to unplug the power of a VM

root@ubuntukvm1:~# virsh destroy vm4
3.9. LIBVIRT VIRTUAL MACHINE MANAGING TOOLS

To do virtual machines migration between host the following requirements must be fulfilled:-

1. Images of virtual machines must be stored in a shared place of both hosts and mounted at same path name on each host. This can be done using iSCSI and NFS.

2. Both hosts must run same type of hypervisors.

3. The necessary ports must be opened.

4. Both hosts must have identical network configuration that supports the migrated host.

**Updating VM’s:** Virsh command is able to change the number of CPUs and Memory size of a VM on the fly. It is also possible attach and detach devices like storage, Ethernet card, USB. but still it is possible to edit the XML file of VMs manually.

```
changing memory and CPU of a VM
```

```
root@ubuntukvm1:~# virsh setmem vm4 524288
root@ubuntukvm1:~# virsh setvcpus vm4 2
```

Virsh uses environment variables that could be seen as a default. For example the variable VIRSH_DEFAULT_CONNECT_URI contains the default hypervisor that could be connected if virsh is used without this option [10].

**3.9.2 virt-clone (command line)**

**Virt-clone** (virt-clone [OPTION]...) is a command line interface for libvirt hypervisors that used to clone(copy) the image of an existing domain and create a new domain. The new domain will have the same resource configurations as the original one, but unique identifiers of a domain will be changed like domain name, disk storage path, domain IP, UUID...[11, 7]

If the necessary information is not available Virt-clone will display error message. But option `--auto-clone`, is used to set the new machine configuration from the cloned one by appending the text `.clon`. For example the name of the clone one is `vm1` the new domain name will be `vm1.clon`. Virt-clone can also have `--prompt` option that used to get necessary information from the user [11, 7].
3.9. LIBVIRT VIRTUAL MACHINE MANAGING TOOLS

Minimum required options are: \texttt{--original} or \texttt{--original-xml} (to specify the guest to clone), \texttt{--name}, and \texttt{-file} (storage path).

--- Cloning VMs using virt-clone command

This example clones a VM called VM1 to VM2 in the local qemu hypervisor. Two file paths have assigned to VM2, because VM1 have had two storage media.

\begin{verbatim}
virt-clone --connect qemu:///system --original VM1 --name VM2 --file /var/lib/qemu/images/VM2a.img --file /var/lib/qemu/images/VM2b.img
\end{verbatim}

For more information see the manual page of virt-clone [11]

3.9.3 virt-install (command line)

\texttt{virt-install} \ (virt-install [OPTION]...O) is a command line tool used for creating and installing a new machine on libvirt hypervisors. It has the features of both a command and graphical base tool [12]. While creating the VM it is possible to configure to use more than virtual disks, network interfaces, audio devices, and physical host devices like USB, PCI etc. Installing the OS could be possible from remote servers using NFS, HTTP, FTP [12, 7].

If virt-install runs with all necessary option, it will do the installation automatically, but it has also a \texttt{--prompt} option that enables intermediate interaction with external users. Virt-install lacks an automated way of creating and installing a number of VMs continuously; that means that creating new VM’s needs human intervention. Basic command options of virt-install are listed below:

--- Creating new VM using Virt-install command

Installing a guest by creating a new storage file, virtual network, boot from the host CDROM, using +VNC server/viewer.

\begin{verbatim}
# virt-install --connect qemu:///system --name VM1 --ram 500
   --disk path=/var/lib/libvirt/images/VM1.img,size=5
   --network network=default,model=virtio
   --vnc --cdrom /dev/cdrom
\end{verbatim}

To get more information see the manual page [12]

3.9.4 virt-(top, what, df)

There are some other virt command line tools like \texttt{virt-what}, \texttt{virt-top}, \texttt{virt-df} which are equivalent to Linux \texttt{what}, \texttt{top} and \texttt{df} commands. Virt-top collects data from a host about the numbers of VM’s in a status( like running, idle, crashed and
3.9. LIBVIRT VIRTUAL MACHINE MANAGING TOOLS

down) *Virt-viewer*, and CPU and Memory usage. *Virt-viewer* is a light weight inter-
face that used to interact with domain operating systems who has graphical user
interface; it uses GTK-VNC technology which is similar to the old VNC service. [7]

### 3.9.5 virt-manager (GUI/command manager)

*virt-manager* is a very simple graphical and command line user interface man-
agement tool for libvirt hypervisors. By clicking on buttons it is possible to create
new domain and install OS, crate and configure networks and storages, cloning
domains. It is very easy to boot up, shutdown, pause, resume, suspend, restore
domains. But it is not possible to manage more domains automatically at a time
[13, 7].

*Virt-manager* also show live resource usage of domains (CPU, Memory and I/O) in
graph form. It has also options in the mine that gives a basic information about
each domain (see figure 3.3).

![Graphical interface of virt-manager tools. By right-clicking on each domain it is possible to start, stop, migrate, clone and configure the domain.](image)

### 3.9.6 Guestfish: interactive manager

*Guestfish* is a command line interactive tool used to modify and manage virtual
machine file systems through libvirt. Using *guestfish* it is possible to play as a
3.10. DRAWBACKS OF EXISTING MANAGEMENT TOOLS

root user can do inside the VM. It is similar to getting the machine directly and running Linux commands such as list files, adding users, controlling file system, etc. Guestfish can edit different types of file systems Ext2/3/4, btrfs, FAT and NTFS, LVM, and any partitions. Guestfish could be used in C, C++, Java, Python, Perl programming languages [1]. The general structure of guestfish is shown follows.

```c
/* this program is used to get access create a
 * file called /hello on an image file called VM1.img
 */
guestfs_h *g = guestfs_create ();
/* adding image file */
guestfs_add_drive (g, "VM1.img");

/* Letting the image file to work */
guestfs_launch (g);

/* Accessing partitions in the image file */
char **partitions = guestfs_list_partitions (g);
char **logvols = guestfs_lvs (g);

/* Mounting the partition to access the file system */
guestfs_mount (g, "/dev/sda1", "/");

/* It is now ready to do any action on the file system
 * the example below creates a hello file */
guestfs_touch (g, "/hello");

/* Saving the action */
guestfs_sync (g);

/* Ending the session of 'g'. */
guestfs_close (g);
```

3.10 Drawbacks of Existing Management Tools

3.10.1 Automation tools

Even though both MLN and RHEV have lots of features, they still have some drawbacks:

Both have a principle of adding VM names into groups to take automatic actions. But this principle increases the management complexity in some organizations those are managing continuously changing VMs, like Universities, Collages, Training Centers. For example, Universities accept/graduate thousands of students each year, students are adjusted to departments, in each department there are different level students. Imagine, every time thousands of VM names must be added into groups, which will create overload to administrators. In addition to that RHEV lacks creating number of VMs automatically and MLN creates new VMs automatically but each group must be written into files based on MLN language.
3.10. DRAWBACKS OF EXISTING MANAGEMENT TOOLS

3.10.2 Libvirt Management Tools

Even though all the above libvirt VM management tools have the power to manage a VM in all aspects but they lack an automated approach. From this point of view this paper will going to develop a new tool that is going to automate VM managements using the above libvirt management tools. The tool will have the capability of creating and managing number of VMs without the need of grouping VMs and configuration files, which will minimize automation complexities and increases usage simplicity.
Chapter 4

"buna": Methodology

The prototype, buna, promises to simplify virtual machine management in all hypervisors that libvirt supports. buna methodology has two parts called automation and plugin part, which are explained in the following sections.

4.1 Automation Part

The methodology of automation part makes virtual machine management simple, flexible and powerful through the help of a naming algorithm. This has a considerable benefit for organizations that require large numbers of virtual machines that change dynamically, like training centers and universities.

University Scenario: Consider a university is managing thousands of students and plans to give Windows7 operating system for all students. Engineering students need Ubuntu-Desktop in addition to the windows7 machines, and Network and System Administration (NSA) department students need both Ubuntu and Windows2010 servers.

Based on the existing automaton tools, each VM must be created either one by one, or all VMs must be listed down in files (called groups) for further process; which is time consuming and becomes a burden for administrators. This thesis suggests to create relationships between virtual machines using their names and then use the relationship to easily manage them.

If a VM has a name related to its purpose, group of VMs with same purpose shall have names with at least one or more similar character. The common substring can then be used to apply an action on a group of VMs.
4.1. AUTOMATION PART

4.1.1 Creating Name Based Relationship Between VMs

It is known in a virtualization environment that:

1. libvirt managers, including all hypervisors are using VM-IDs or VM-Names to access virtual machines (sec. 3.8.1.1); for this \textit{buna} follows using the names of virtual machines for management.

2. giving \textit{meaningful names for VMs} is not difficult as long as they are created for certain purposes. This can be done based on the virtual machines’ purpose, resources type and operating system they are using. For example the name of a Windows7 virtual machine that is created for a first year Biology department student can be ‘Win7-1st-VM1’.

While giving meaningful name if some care is taken on the substring, like \textit{grouping} and \textit{ordering} them, to formulate a mathematical relationship between the names, then it is possible to access a number of VMs through their relationships (common substring).

From the University scenario, the meaningful names for VMs can be given based on the tree structure shown on figure 4.1. Hence, the \textit{Biology} department virtual machines can be accessed using the common substring ‘Bio’; also to get all \textit{Windows7} virtual machines one can use the substring ‘Win7’. This creates an environment to manage thousands of VMs in a simple way.
4.1. AUTOMATION PART

Figure 4.1: Generating a meaningful name for a VM can be done by concatenating all/some substring existing through a path beginning from the root to each leaf node.

4.1.2 Generating Meaningful Names

Once the relationship tree structure is formulated, by categorizing all substrings into sets and then by applying the one direction cross-product between subsets of these sets, meaningful names for virtual machines can be generated. The available sets from the University scenario tree structure are listed below.

<table>
<thead>
<tr>
<th>Available sets from University scenario</th>
</tr>
</thead>
<tbody>
<tr>
<td>same level elements are listed in a set :</td>
</tr>
<tr>
<td>OS = {UbuS10, WinS10, UbuD, Win7}</td>
</tr>
<tr>
<td>department = {NSA, Eng, Bio, ...}</td>
</tr>
<tr>
<td>year={1st, 2nd, 3rd, 4th, 5th, ...}</td>
</tr>
<tr>
<td>postfix = {vm}</td>
</tr>
<tr>
<td>number ={1, 2, 3, ..., N}</td>
</tr>
</tbody>
</table>

The one directional cross product between sets, to generate meaningful names of virtual machines, can be done as follows.

<table>
<thead>
<tr>
<th>Sample for one direction cross-product for NSA department</th>
</tr>
</thead>
<tbody>
<tr>
<td>{UbuS10, WinS10} X (NSA) X {1st, 2nd} X {vm} X {1, 2, 3, ..., N}</td>
</tr>
</tbody>
</table>

Result of one directional cross product looks:

{(UbuS10, NSA, vm1), ..., (WinS10, NSA, vmN)}
4.1. AUTOMATION PART

Meaning full names can be generated as:
UbuS10-NSA-1st-vm1, ..., WinS10-NSA-2nd-vmN.

Once virtual machines are assigned these names, it is possible to access them using
the wild-character, '*', and take automatic actions using libvirt managers like virsh
and virt-clone. E.g. to refer all virtual machines existing in the NSA department,
one can use *NSA*. Similarly to get access only Ubuntu server virtual machines,
one can use UbuS10*.

4.1.3 "buna" Naming Convention

To implement this naming method in a simple way, buna imposes the following
syntax rules as buna convention on its users.

1. A name should contain alphabets, digits and special characters "," and ".";
   all other characters are not considered and shall be escaped if they are used.

2. Elements in a set must be separated by space and listed after opening curl-
   brace '{' (there is no need to put closing curl-brace '}' at the end). This will
   make the name listing simpler. E.g. {Win2010-server Ubuntu-server {vm1
   vm2 vm3 vm4 {1st-year

3. Sequential numbers can be expressed using two dots without space. ex
   vm{1..20 or vm1..20 =⇒ generates vm1, vm2, vm3 upto vm20

4. Cross-product will be applied between a series of listed sets until the first
   closing curly-brace is encountered. All sets listed after the closing brace will
   be treated as a new name pattern. e.g. WinS10 UbuS10{-NSA-{vm1....20}Win7-
   {Bio Chem Phy{vm1..10 =⇒ this expression has two name patterns. The
   first one is WinS10 UbuS10{-NSA-{vm1....20 and the second one is Win7-
   {Bio Chem Phys{vm1..10. Applying cross product on the first name pattern
   generates names as WinS10-NSA-vm1, ..., WinS10-NSA-vm20, UbuS10-NSA-
   vm20, ..., UbuS10-NSA-vm20, and the second pattern generates names as
   Win7-Bio-vm1, ..., Win7-Chem-vm1, ..., Win7-Phy-vm10.

5. Add period "." at the end of an element to terminate further forward cross-
   products through that element. e.g. Win7{- assistant. Bio {-vm1..2 =⇒
generates Win7-assistant, Win7-Bio-vm1 and Win7-Bio-vm2. The sub-string
'assistant' is exempted from further cross-products with the set {-vm1..2 }

6. Use the wild-character '*.' to represent any substring on VMs’ names.
4.1. AUTOMATION PART

7. When new VMs are created, the template-VM-names must be given by using colon(;) next to a group element. e.g. WinS10:WinServer-template|-NSA-{vm1..20} ➞ create VMs like WinS10-NSA-vm1, ... WinS10-NSA-vm20 using the template VM called "WinServer-template".

For more examples of naming expressions, see appendix ’B’.

If virtual machine names are given to buna by following the above rules, buna will expand, generate, polish and filter the input to get the right names of virtual machines; and then takes any given action on each virtual machine with the support of libvirt managers such as virsh and virt-clone. The next chapter explains in detail the design of buna following this methodology.
4.2 Plugin Part

This section shows the method for managing resources of hosts and controlling the status of VMs based on the automation part.

4.2.1 Methods of Managing Hosts

Consider there are two hosts named \( H1 \) and \( H2 \) that are connected to each other and have a number of virtual machines managed by libvirt.

A host has resources usage like \( CPU \), \( Memory \) and \( Network \); all VMs residing in the host are in a particular state like \( active \), \( inactive \), \( running \), \( blocked \) and \( paused \). Let’s name both these resources and states as \textbf{properties}.

Using \textit{virt-top}, collect data about the properties from each host with in a given time interval and sample size. Once the data is collected, perform some descriptive statistics (like mean, mode and median) about each property of each host. On the other hand, allow the user to insert \textit{rules}, where each rule has a \textit{condition} and an \textit{action} part. The condition part is a logic and numeric based expression related to the statistics. Lastly evaluate the \textit{rule-condition} part against the collected property statistics: if the \textit{condition} is fulfilled, then execute the \textit{action} part. The action part can be \textbf{buna automation} or any Linux command, program and/or script.

To implement this idea, \textbf{buna} has a program named \textit{skeleton} and a file named \textit{rules-file} which are described below.

4.2.2 Skeleton Program

The skeleton is the main program that is copied into a daemon file with some modifications on variables and if-condition-action codes from rules-file. It will then execute as a Linux daemon process.

Basically the skeleton program has three blocks of codes named \textit{variable-block}, \textit{statistics-block} and \textit{rules-block}. The \textit{variable-block} contains a list of global variables that are used to represent host-names, host properties and statistics parameters. Some of the variable values will be modified based on the rules-file. The \textit{statistics-block} is not a user modifiable part. Within a given time interval and sample size, it collects data and generates statistics about the properties of each host. The \textit{rules-block} of the skeleton is a free space that can be filled with if-condition-action codes translated from the rules-file. Then the codes in the rules-block will be executed after the periodic statistics calculation is completed (fig. 4.2).
4.2. PLUGIN PART

In short, when `buna` runs as a plugin, the following steps will be processed:

1. Copy the skeleton program to a new program file that is given by the user, which is referred as `daemon-plugin`.

2. Update the variable values based on the rules from the rules-file. E.g., if H1 and H2 are used in the rule condition part, they will be included in the host variable; and statistics and actions will be done on them. Rules will be discussed latter (sec. 4.2.2).

3. Convert each rule from rules-file into *if-condition-action* code and write into the rules-block part of the daemon-plugin file.

4. Finally execute the `daemon-plugin` as a Linux process (fig. 4.2).

![Diagram](image)

Figure 4.2: The figure shows the approach to manage resources between hosts in an efficient way. The dotted-square-area shows the skeleton program while it is modified with rules-file to generate the complete daemon-plugin.

**Statistics Variable**

The statistics that is done for hosts of each property will be stored in a variable, where the name of the variables have a format: `[Host-Name].[property].[statistic]`. 
4.2. PLUGIN PART

Having this similar format for all variable names makes the way of referring their value through rules file simpler.

---

The presentation format for host property statistics result.

```
[Host-Name].[property].[statistic]
```

(H1 H2).{memory cpu running idle crashed}.{mean mode percentile median min max}

- H1.running.max =Y
- H1.guestmemory.percentile =X
- H2.cpuusage.mean=Z
- H2.active.mean=A

---

It’s a file created by the user that contains some global variables and a number of rules that are written based on *buna-rule-syntax*. Host names that are used in the rules file will be filtered, evaluated and then added into the variable-block of the copied skeleton program. Then each rule will be converted into if-condition-action code and written into the rules-block of the copied skeleton program. After this, the complete daemon plugin program is formulated and executed.

### buna-Rules-Syntax

Rules are written by using already existing *hosts*, which are listed in *buna.conf* file, and following *buna-rules-syntax*:

Each rule has to be written as $\textit{rule}(\textit{CONDITION}; \textit{ACTION}; \textit{FLAG})$. The CONDITION is written by using the mathematical logic (AND/OR) & inequality symbols (=, >, <). The value between the inequality symbols must be an integer or a variable written as $<\textit{host-name}>.<\textit{property-name}>.<\textit{statistics-name}>$. Anything included in the ACTION part will be executed when the CONDITION part is fulfilled. The ACTION part can be a *buna* automation command or any Linux command, program and/or script. An example on how to write rules is shown below:

---

### Sample rules file structure

```
$interval = 120; # time interval in second to collect each sample
$percent = 75; # number of samples going to be collected

----- Example user defined

$rule(H1.running.max <= 4; buna --action start --vm backup-vm4);

$rule(H1.running.max > 4; buna --action destroy --vm backup-vm4);

$rule(H2.running.mean > 10 or H1.running.mean > 8; echo overload!!!!!!);

$rule(H2.running.max > 10 and H1.running.max < 3; buna --action migrate
  --source remote --dest local -- --vm running --count 3);

$rule(H1.memory.mean < 150%; start a backup program or VM);
```
4.2. PLUGIN PART

Variables that are generated by rules-block of the skeleton program and used in the rules file must be similar. This is because it will make the rule translation from the rules-file to the rule-code simpler.

Based on this approach, buna can control resources and virtual machines as per the needs of users. Writing rules is user friendly and flexible enough. Also rules can be constructed do effective tasks like load balancing and power saving. The plugin part of buna design (sec. 5.4) explains it in detail.

Alternative Approach

There is also another approach for the plugin part: Every time after collecting and doing the statistics; read each rule in the rules-file and take actions by evaluating the given conditions. But this will affect the performance of the system because the file will be opened and closed every time; and it has a risk if the rule-file is missed, corrupted or changed.
Chapter 5

"buna": Design

This chapter discusses the design of buna, which follows the method stated in the previous chapter. The explanation starts from high level structure of buna that shows its interaction with external users and libvirt, and then it goes deep down to each individual process that is used in the automation and plugin parts. The explanations are supported by DFD, Flowcharts, tables and examples.

5.1 Context Diagram

buna is a command line tool that creates a communication environment between users and libvirt APIs. buna gets input values through the following three sources:

1. Input options: buna has numbers of input options those can be used while running it as a Linux command.

2. Configuration file called buna.conf.

3. From rules those are written in a file called rules-file.

By using the values from the above three sources, buna interacts with libvirt APIs to automatically manage virtual machines on different hypervisors. If any error is occurred, while using the above inputs, buna replies to users with errors and additional information (DFD 5.1).
5.1. CONTEXT DIAGRAM

Figure 5.1: Context Diagram of \texttt{buna}. \texttt{buna} interacts with external user directly with command options or indirectly through storage files, and interact with libvirt APIs through commands.

5.1.1 Input Options

As it is common in many command line tools, \texttt{buna} also has number of options that are used for both \textit{automation} and \textit{plugin} parts.

```plaintext
```

From these options, \texttt{--plugin} is used for the \textit{plugin} part of \texttt{buna}; while the rest are used for the \textit{automation} part. The functionalities of these options are described bellow:

\textbf{\texttt{--information <command>}}: can be shortened as \texttt{-inf}. This option will take \textit{virsh manager commands} that allow users to get information about VMs on a host. It works together with \texttt{--vm}, \texttt{--connect} and \texttt{--count} options. When these options are fulfilled, \texttt{buna} formulates a virsh command that is used to gather information and executes it on libvirt APIs automatically (sec. 5.3.1). The available commands of information are listed below.

- \texttt{running} : to see all running VMs in a specified host
5.1. CONTEXT DIAGRAM

• idle: to see all idle VMs
• paused: to see all paused VMs
• down: to see all down VMs
• off: to see all switched off VM
• crashed: to see all not working VMs
• dying: to see VMs that are on the process of shutting down
• all: to see the status of all VMs
• dominfo: to get information about each VM that includes: the VM id, name, running operating system, current state, number of assigned CPUs, Max and current memory size, security model,...

• domuuid: to get VM’s global domain identification number
• domid: to get the temporary identification number given for running VMs
• vcpiinfo: to get the number of virtual CPUs used by each VM.
• ttyconsole, vncdisplay : to see the type of console and VNC display used by each VM.

--action <command>: can be shortened as -act. This option takes selected commands working in virsh and virt-clone. Commands are those used for doing some action on VMs like start, stop and clone. The ‘--action’ command works with --vm, --connect, and --count options. When these options are fulfilled, butna will generate and execute an automatic virsh manager command on libvirt APIs (sec. 5.3.2). The available commands of --action option are stated below.

• Start, destroy, suspend, resume, reboot, shutdown, undefined: all these command values are used to take action according to their dictionary meanings. For example start value is used to start all given VMs in all given hosts.
• setmem setmaxmem setvcpus : all this command values are used to set the current or maximum memory size in kilobyte, assign number of virtual CPUs for all VMs given by –vm option.
• rename : is used to change names of virtual machines and it has great support for regrouping virtual machines. It works with ‘--change’ option (sec. 5.3.2.2).
• clone: is used to duplicate template VM to new VMs given by --vm option.
5.1. CONTEXT DIAGRAM

- **migrate**: is used to take CPU and memory snapshot running VMs given by \(--vm\) option from one host (given as \(--connect\)) to the other (given as \(--destination\)). This functionality has great support to implement load balancing and power saving operations.

- **save, restore**: these options are used to save running VMs or to restore and use saved snapshot of VMs.

\(--device[<command><values>]\): can be shortened as \(-dev\), This option takes an action to attach or detach devices to/from VMs. The value of this option is a virsh device command with its value. The values of these commands are either a device-XML-file (to attach the device) or device-name (to detach the device). When these options are fulfilled, buna will generate and execute an automatic virsh manager command on libvirt APIs (sec. 5.3.3). The available options are listed below.

- attach-device [FILE-xml-of-the-device]
- detach-device [DEVICE-FILE]
- attach-disk [SOURCE-file-access-path] [TARGET-device-access-path]
- attach-disk [TARGET-of-the-device]
- attach-interface [TYPE-network-or-bridge] [SOURCE-device]
- attach-interface [TYPE-network-or-bridge]

see virsh manual[10] for detail understanding.

\(--connect <Host-list>:\) is used in all virsh, virt-clone and virt-top managers, and that is also used in buna. It can be shortened as \(-conn\). It inputs list of hosts that are going to be targeted by information, action, and device processes.

\(--vm <VM-Name-syntax>:\) it takes an expression written based on buna-naming-convention (sec. 4.1.3). The expression will be processed to generate the right VM name and get ready for information, action and device processes (sec. 5.2.3).

\(--directory <full-path>:\) can be shortened as \(-dir\), it is a directory path where the memory and CPU snapshot of running VMs are going to be stored. This option will be used during \(--action save\) and \(--action resume\) action values.
5.1. CONTEXT DIAGRAM

–destination <dest-host>: can be shortened as -des, it is the host name of a destination hypervisor where VMs are going to migrate into. This option will be used with ’–action migrate’ option.

–count [<memory-size | number-of-cpu | number-of-VMs]: the value of –count can be memory size in kilobytes, number of CPUs going to be assigned to VMs. Or the number of random VMs going to be selected from a state. This has a benefit for different operations like migrating any running VMs from one host to another.

[–change <old-substring-of-VMs-name> <new-string-for-VMs-name>]: This option is used when renaming virtual machines. The first input is a substring from existing virtual machine names and the second one is a new string that will replace the first one. This option works with ’–action rename’ (sec. 5.3.2.2).

–live, –remove –help: –live is used to migrate VMs without interrupting them from running, this option works with ’–action migrate’. –remove is used for delete image files while VMs are undefined from the system; this option works together with ’–action undefined’. By default image files are not removed even if VMs are undefined. Option –help is used to see buna usage.

5.1.2 Configuration File (buna.conf)

this is a configuration file used to change the values of some global variables. At the initial stage of running buna, it reads the file to change the global variable values. The available variables which can be changed using buna.conf are:

1. $Host: it is used to list all available hosts with their name and hypervisors’ uri. So that host hypervisors are referred using their name through buna with option like –connect host-name. Any host not listed in buna.conf will not be considered for further process.

2. $USER: it is a common ssh user for all hosts, default is root.

3. $FILE_DIR: it lists the available disk paths that are ready to store image files of new VMs. Storages are listed here based on priority, the higher priority path will be used first and then if this storage becomes full the next storage directory will be used. Any storage path that is not included in buna.conf will not be considered.

4. $DIRECTORY: it is used to give a directory path that is going to be used for storing CPU and Memory snapshots of running VMs.
5.2. HIGH LEVEL STRUCTURE

5. $XML$: it is a directory path where the virtual machines of XML file are located. The XML file of VMs will be referred here when renaming VMs.

Syntax of writing on *buna.conf* looks like as follows (appendix 'C'):

```sh
# --> $HOST(alayses-name, uri);
$HOST=(local, qemu:///system);
$HOST = (remote, qemu+ssh://192.168.0.27/system);

# --> $FILE_DIR = full_file_path, priority;
$FILE_DIR=/home/tow/dev/rule,3;
$FILE_DIR=/var/lib/libvirt/images/iscsi,1;

# --> $DIRECTORY = full_directory_path
# default is /var/lib/libvirt/store/
$DIRECTORY= /your/full/storage/directory/path;
```

5.1.3 Rules File

It is a file created by users and contains list of user defined rules, which are written in accordance with *buna-rule-syntax* (sec. 4.2.2) and feed to *buna* using *–plugin* option. The file also used to modify some global variables while a plugin-process is generated and executed as a Linux daemon (sec. 5.2.5).

5.2 High Level Structure

All options’ values given by users will be analysed through the first process called *Option-Manager* and then further process will be done in one of the processes called information, action, device and plugin (DFD 5.2).

5.2.1 Option-Manager

It is responsible to validate inputs, change global variables using *buna.conf*, translate the given host names to their *uri*, and generate full VMs’ names given by *–vm* option. These all processed values will be feed to one of the processes: *information, action, device* and *plugin*. During the process, any information or error will be replied to the user. To fulfil all these tasks, *Option-Manager* has *sub-processes* stated below.
5.2. HIGH LEVEL STRUCTURE

Figure 5.2: High level structure of *buna*, input options will be sensed by sub-processes and forwarded for further processes if they are free from error. Further processes are plugin, information, automation and device.

**Grouping Option Values**

*Buna* listed down all available option values into groups named STATE, INFO, EVENT_ACTION, UPDATE_ACTION, DEVICE_ACTION, ACT_INACT. All commands existing in *virsh*, *virt-clone* and new *buna* values are categorized into the above groups. Values in the same group are those that need same type of additional options during execution time. For example to start, stop, pause or destroy a VM using *virsh* manager, the command looks like "*virsh [start | stop | pause | destroy ] <vm-name-here>*". So listing these commands in the same group and writing a code that represents each group will simplify the management system. Each group values are shown on table 5.1.

Most of the values listed in the last column are *virsh* and *virt-clone* commands. In the STATE group, there is a value named "*all*", it is used to represent all values in the group. Values listed in ACT_INACT will be used for --vm option that is used to refer all, active or inactive VMs. E.g. if a user need to stop all active VMs, "--vm active and --action destroy" options will be combined.
5.2. HIGH LEVEL STRUCTURE

Table 5.1: Grouping *buna* option values.

<table>
<thead>
<tr>
<th>Group</th>
<th>Description (used option)</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>@STATE</td>
<td>get VMs with their statuses (--info)</td>
<td>running, idle, paused, down, off, crashed, dying, all</td>
</tr>
<tr>
<td>@INFO</td>
<td>detail information about VMs (--info)</td>
<td>dominfo, domuuid, domid, vcpuinfo, ttyconsole, vncdisplay</td>
</tr>
<tr>
<td>@DEVICE_ACTION</td>
<td>device attach/detach to VMs (--device)</td>
<td>'attach-disk', 'attach-interface', 'detach-device', 'detach-disk', 'detachinterface'</td>
</tr>
<tr>
<td>@EVENT_ACTION</td>
<td>basic actions on VMs (--action)</td>
<td>clone, start, destroy, suspend, resume, reboot, 'shutdown', save, restore, migrate, undefine</td>
</tr>
<tr>
<td>@UPDATE_ACTION</td>
<td>updating CPU and Memory (--action)</td>
<td>setmem, setmaxmem, setvcpus</td>
</tr>
<tr>
<td>@ACT_INACT</td>
<td>referring VMs in their status (--vm)</td>
<td>all, active, inactive</td>
</tr>
</tbody>
</table>

All input options given to *buna* will be checked in which group it is in and then forwarded to the corresponding process (see 5.2.5).

5.2.2 Global-Variable-Updater

This process is used to update the values of global variables if there is a change in *buna.conf*. While reading this file, if there is a declaration like `$HOST=(hostname, uri)`, the host-name and uri will be arranged as key and value combinations and feed to %HOSTS global variable. If there is a declaration like `$FILE_DIR=VM-image-store-dir, priority-number)`, then it will be feed to @FILE_DIR and the directory that is used for storing snapshots will be assigned to $DIRECTORY (Flowchart 5.3). Then after, these variables are ready to be referred by other processes (Flowchart 5.3).
5.2. HIGH LEVEL STRUCTURE

Figure 5.3: Flowchart for updating global variables like `host-uri`, `VMs image` and `snapshot storages` path through `buna.conf` configuration file.

5.2.3 Name-Generator

This process is called when `buna` runs with `–vm <vm-name-syntax>` option. It extracts the syntax and generates full names of the targeted VMs and then for further investigation it forwards the names to `Name-Filter`. The procedure of extracting the given syntax is done as follows (Flowchart 5.4):

To make it clear, consider a user needs to create new VMs using a `vm-name-syntax` like: `–vm win2010 ubuntu-server{-NSA-{assistant vm1..5}win7-{john pitter}{-vm1 -vm2`.

1. By using the *closing curl brackets*, separate the *string* that represents each...
5.2. HIGH LEVEL STRUCTURE

cross product area from the given *vm-name-syntax*. Any substring before or after the closing curl bracket is considered as a set area that is going to be used for cross product. The example has two cross product area: ‘`win2010 ubuntu-server|-NSA-{assistant vm1..5}`’ and ‘`win7-{john pitter}{-vm1 -vm2}`’.

2. By using the *opening curl brackets*, separate the substring of each set existing in the cross product area. At this time each new substring represents a set of a cross products. From the previous step, the first cross product area substring will be divided into string sets as ’`win2010 ubuntu-server`’, ’`-NSA-`’ and ’`assistant vm1..5`’. The second cross product substring will be divided into string sets as ’`win7-`’, ’`john pitter`’, and ’`-vm1 -vm2`’.

3. Expand the substring of each set if continuous numbering is shown through the substring. From the previous step the set substring ’`assistant vm1..5`’ will be extracted to a new set substring like ’`assistant vm1 vm2 vm3 vm4 vm5`’.

4. Collect all elements of each set in a cross product area, which is done by splitting each set substring using *space character* (because elements of a set is separated by space). From the previous step, elements of the sets in the first cross product area are: @set1={win2010 ubuntu-server}, @set2={-NSA-}, @set3={-assistant, vm1, vm2, vm3, vm4, vm5}. The second cross product area string looks like @set1={win7-}, @set2={john, pitter} and @set3={-vm1, -vm2}.

5. Generate the full name by applying the forward cross product rule between sets in the same cross product area. The first cross product area generates full VMs’ names in the order of: *win2010-NSA-assistant, win2010-NSA-vm1, ..., ubuntu-server-NSA-assistant, ... ubuntu-server-NSA-5*. Similarly the second cross product area will generate full names like *win7-john-vm1, win7-john-vm2, win7-pitter-vm1, win7-pitter-vm1* in the same generate the VMs full name.

6. Finally give the whole generated names to *Name-Filter*, that filters and polishes the names (Flowchart 5.4).
Figure 5.4: Flowchart of Name-Generator: generates the full names of VMs from the input given through \texttt{-vm} and feeds the result to \textit{Name-Filter}.
5.2. HIGH LEVEL STRUCTURE

5.2.4 Name-Filter

This process accepts the extracted VMs’ names from Name-Generator and does filtering and some polishing on the VMs’ names based on the value given through \textit{--action}. The steps used for filtering names are shown below:

1. If names are going to be used for creating new VMs, then it will remove characters which are not allowed to use for naming. From the above example, if the substring ’\textit{win7}’ is written as ’\textit{win;7}’, or ’\textit{win/7}’, then \textit{buna} will automatically remove the wrong characters and takes the properly used parts (in this case ’\textit{win7}’).

2. If the generated names are used for referring the existing VMs in a host, then it filters out the names of requested VMs from hosts using wild character ’\texttt{*}’. \textit{buna} uses ’\texttt{*}’ for string matching in a similar way as it is used in most systems like Linux shell and SQL (E.g. ’\texttt{--vm *server*}’).

3. \textit{Buna} has some special keywords called ’\texttt{all}’, ’\texttt{active}’, and ’\texttt{inactive}’, which can be a value of ’\texttt{--vm}’. These keywords are used to list all VMs, active VMs and inactive VMs respectively. For example, if a user runs \textit{buna} by ’\texttt{--vm inactive}’ option, then ’\texttt{virsh --connect $uri --inactive}’ will be executed. From the result of this execution, all virtual machines which are in the generated VMs’ name list will be filtered.

4. Finally it returns the filtered VMs’ name to the main processor called Option-Manager (Flowchart 5.5).
Figure 5.5: Flowchart of Name-Filter: filters the existing VMs from hosts based on the names collected from Name-Generator and then returns them to Option-Manager.
5.2. HIGH LEVEL STRUCTURE

5.2.5 Type-Checker

This sub-process of Option-Manager decides which process is going to be triggered to accomplish the user task; and then forwards both the uri and filtered names to the selected process (Flowchart 5.6). The available processes are information, action, device and plugin. The decision of selecting a process is done by the following input options taken from users:

1. –plugin <value>: calls plugin process to be used
2. –information <value>: calls information process to be used
3. –action <value>: calls action process to be used
4. –device <value>: calls device process to be used

From the above, only ’plugin’ processes is used for plugin part. The rest three processes (i.e. information, automation and device) are all used for automation part. The next section will give detail explanation about these four processes categorized as Automation and Plugin part (Flowchart 5.6).
5.2. HIGH LEVEL STRUCTURE

Figure 5.6: Flowchart for Type-Checker process: It decides which process is going to be triggered to process the users task.
5.3 Automation Part

The automation part of *buna* is designed to do any automatic events on VMs within a simple, flexible and powerful way. The options are –information, –action and –device. Most of the values of these options are directly taken from *virsh* and *virt-clone* managers (sec. 5.2.1), so that users who are already familiar with these managers will not get difficulty to use buna. Any automatic event ordered by users is processed with one of the processes: information, action, device and plugin.

5.3.1 Information Process

When *buna* runs with –information <command> option, the Information process will be selected to give information about the selected VMs from Name-Filter process. The information option runs with the combination of –connect, –vm and –count options. When this options are fulfilled, the Information-Automation process will generate *virsh* command using the name of each VM and execute it on libvirt APIs (Fig. 5.7).

![Diagram of information-automation process](image)

**Figure 5.7: DFD of Information-automation:** It collects information about VMs through virsh command and returns the result to users.

**Information-Automation**

All input values (the filtered VMs’ names (sec. 5.2.4), uri and the value given to –information (sec. 5.2.1)) will be given as an input to Information-Automation. If the information value is ‘all’, then *virsh list –all* command will be executed and the result will be returned to the user. If the value is an element of @STATE, then VMs with the given status will be listed. If the value is in @INFO elements, detail information about each VM will be collected (Flowchart 5.8).
Figure 5.8: Flowchart of Information-Automation: That is used to gather information about VMs based on the values given for –information and –vm options.
5.3.2 Action Process

This process will be selected when users run `buna` with option `--action <command>`. It’s designed to take action on the given VMs and hosts by orchestrating `virsh` and `virt-clone` manager. Any information and error generated during the process will be forwarded to the user (see image 5.9).

![DFD of Action-Automation process](image)

Figure 5.9: DFD of Action-Automation process, it collects all names of VMs and apply the given action on each VMs by using virsh/virt-clone commands.

The available actions in this process are categorized as `Event_Action` & `Update_Action` (sec. 5.2.1). From all actions, ‘clone’ & ‘rename’ are a very basic actions that need to be seen first.

### 5.3.2.1 Clone-Action

This action is used to create Vms by cloning the template VMs. Cloning needs a lot of considerations like generating VM names, filtering the assigned template VMs and cross checking the free storage space against the size of image files of new VMs. From this point of View a new function named `Clone-Function` is developed.

The `Clone-Function` accepts filtered names from `Name-filter`, gets template VMs from `–vm`, collects available storage path from `buna.conf` and the flag `–live` (for live migration). The function orchestrates all these inputs to generate and execute the `virt-clone` command for each virtual machine (Flowchart 5.10). The steps of this function are:

1. Filter out all templates VMs’ given from `–vm` input, where template names are included as "group-string:template-vm-name" (sec. 4.1.3).
2. Check whether the available storage space is enough to store images of new virtual machines. The user will be informed if the space is not enough.
3. Finally for each virtual machine a `virt-clone` command will be generated and execute on libvirt API.
Figure 5.10: Flowchart of Cloning-Automation process, it generates a virt-clone command for each VM name and runs on libvirt API.
5.3. AUTOMATION PART

5.3.2.2 Rename-Action

Changing virtual machine names can be done by modifying their XML file. Most of the time the images of virtual machines are saved with their names, so that renaming their image file name and modifying the link path in their XML file is necessary. `buna` options that are used for renaming VMs are `--action 'rename'`, `--vm <vm-name-syntax>` and `--change <old-substring-of-VM-name> <new-string-for-VM-name>`. First `buna` gets the targeted virtual machine names from `Name-Filter` process, then renames each virtual machine as follows:

1. The new virtual machine name is formulated by replacing a substring from the old VM name with a new string, strings are taken from `--change` option.

2. Open the virtual machine XML file using `virsh --dump` command, and change the old name with the new name on the line that contains `<name>new-VM-name </name>`. The line that contains its image path shall also be changed with the new name. After doing that save the modified XML file with the new name and delete the old XML file.

3. Finally, change the name of image file of the VM to the new one as it is modified in the XML file.

After applying the above modification for each virtual machine, restart `libvirt` daemon to refresh the system (Flow-chart 5.11).

Figure 5.11: Flowchart of Rename-Automation: It is used to change the names of virtual machines, which has a grate impact for regrouping VMs.
5.3. AUTOMATION PART

5.3.2.3 Event-Action


```
# To clone a VM:
virt-clone --connect [ URI ] --original [ $original ] --name [ $vm ] --file [ $file_path ]
# here original is the template virtual machine name
# To pause(suspend a VM):
virsh migrate --suspend $vm [ $DESTINATION_URI ] [ $SOURCE_URI ];
# To save the snapshot of a VM in $snapshot-path
my $command = 'virsh --connect $uri save $vm $snapshot-path;
# To restore a VM from already saved snapshot:
virsh --connect $uri restore $snapshot-storage-path;
# Other actions named $action=start, destroy, suspend, resume, reboot, 'shutdown'
virsh --connect $uri $action $vm;
```

From this point of view five different functions named Clone, Suspend, Save, Restore and other are developed. Each function has a simple loop that runs virsh/virt-clone command with the name of each VM (Flowchart 5.12).

5.3.2.4 Update-Action

Includes: ‘setmem’, ‘setmaxmem’, ‘setvcpus’, and ‘rename’. All of them, except ‘rename’, will be done by using virsh manager with the same input options. The value ‘setname’, included by buna, is used to change the name of virtual machines (Flowchart 5.11). By running the below command with a simple loop for each VM, automatic updating on CPU and Memory can be done (Flowchart 5.12).

```
# To change the CPU and Memory size of a VM, $action= setmem, setmaxmem, setvcpus
my $command = 'virsh --connect ' . "$CONNECT_URI $action $vm $count";
# Here count is the number of CPU or memory size in kilobyte
```
5.3. AUTOMATION PART

Figure 5.12: Flowchart of Action-Automation: by using the value given through 'automation' option, the appropriate virsh action command will be generated for each VM and executed on libvirt API.
5.3. AUTOMATION PART

5.3.3 Device Process

This process is called when `buna` runs with `-device <option>`. This option takes actions to attach or detach devices (like USB and printer) to the selected VMs. The `Device-automation` orchestrates the `virsh` command with the name of each virtual machine and executes it on libvirt APIs (Fig. 5.13). The device that is going to be attached must be a libvirt XML file that contains information about the device. To detach a device from a virtual machine, it needs to give only the device name.

![Diagram of Device-automation process]

Figure 5.13: DFD of Device-automation process: It automates the `virsh` command to attach/detach devices on each virtual machine.

Virsh commands that is used to attach or detach devices on virtual machines are shown below.

```
pattern: virsh [device-command] [vm-name] [other-options]

# To attach/detach other devices:
virsh attach-device $vm-name [xml-of-the-device];
virsh detach-device $vm-name [device-file-path];

# To attach/detach disks
virsh attach-disk $vm-name [SOURCE-file-access-path] \
[TARGET-device-access-path];

# To attach/detach interface:
virsh attach-disk $vm-name [TARGET-of-the-device];
virsh attach-interface $vm-name [TYPE-network-or-bridge] [SOURCE-device];
virsh detach-interface $vm-name [TYPE-network-or-bridge];
```

As shown on the above commands, it is difficult to formulate a common pattern that can be repeated for each VM; but the virtual machine name is always positioned at the second place. To add a simple automatic action on it, the better way is to allow users to insert all the necessary values (except the VM name) as they are used in virsh manager, and give VM names through `--vm` option.
5.3. AUTOMATION PART

the Device-Automation will automatically insert the name of each virtual machine at the second place into the value given through the –device option and executes it using virsh command (Flow-chart 5.14).

Figure 5.14: Flow chart for Device-Automation: that shows a way to attach/detach devices on virtual machines.
5.4 Plugin Part

The objective of buna plugin part is to manage resources between hosts and control status of VMs (sec. 4.2). When buna runs with an input option –plugin <daemon-name rules-file>, the plugin-process will be invoked and it starts writing a new daemon-plugin program. Writing the daemon-plugin is done by copying the skeleton program and updating it as per the rules-file; and finally it is executed as a Linux daemon. Once the daemon-plugin is executed, it starts collecting data using virt-top manager, does descriptive statistics about each host properties, and perform the given actions based on the fulfilment of some conditions that are based on values from the statistics. The daemon will do these tasks periodically and forever unless it is killed by the user itself (DFD 5.15).

Figure 5.15: DFD of plugin-process: A daemon-plugin program is generated by combining the skeleton program and rules-file. The program will run as a Linux process that is used to control resources between hosts.

Detail design of skeleton and plugin-process programs are explained and discussed in the next section.
5.4. PLUGIN PART

5.4.1 Skeleton Program

Skeleton is the steam program that will be copied for all daemon-plugins. The skeleton by itself can’t do anything, unless some additional variables and if-condition-action codes are included from the rules in rules-file. For this, the skeleton program has three main blocks of codes named variable, statistics and rules (Flowchart 5.16).

5.4.1.1 Variable-Block

The variable block of skeleton program has defined global variables, and some of them needs a value from rules-file. These variables are grouped as host and rule based variables.

⇒ Host Property Variables: are global and unchangeable variables that is used to collect statistics (mean, mode, median,...) results about hosts (DFD 5.16).

1. %count: collects the number of VMs existing on a host
2. %running: collects the number of running VMs on a host
3. %blocked: collects the number of blocked VMs on a host
4. %paused: collects the number of paused VMs on a host
5. %shutdown: collects the number of shutdown VMs on a host
6. %shutoff: collects the number of shutoff VMs on a host
7. %crashed: collects the number of crashed VMs on a host
8. %active: collects the number of active VMs on a host
9. %inactive: collects the number of inactive VMs on a host
10. %cpuusage: The percentage of host CPU usage
11. %guestmemory: The percentage of host Memory usage

⇒ Rules-file Variables: These variables are collected from rules file and added into the skeleton program when a daemon-plugin is generated (sec. 5.4.2.1):

1. %hostlist: it contains the list of hosts (host-name,uri), which are going to be managed. Host names are added to this variable if and only if they are used in the rules file and they are also defined in buna.conf.
5.4. PLUGIN PART

2. $samplesize=<Number>$ : this variable contains the number of samples going to be collected to do a periodic statistics about each host properties listed above.

3. $interval=<seconds>$ : the sleep time interval between each sample collection.

4. $percentile$ : this value is used to calculate the percentile of the collected sample.

*If $samplesize$, $interval$ and $percentile$ values are not set in rules-file, plugin-process will define them with some default values.*

### 5.4.1.2 Statistics-Block

The statistics-block of the skeleton program is designed to collect data about each host properties through *virt-top* manager, and perform descriptive statistics about the properties. All these tasks are done with the limited sample size and time interval, which are given through *buna.rules*. The descriptive statistics which are done by this program are stated below (DFD 5.16).

1. *mean*: the average value of the sample collected about each property

2. *median*: the middle value from the sorted sample that are collected about each property

3. *mode*: the highly repeated value from the sample collected about each property

4. *percentile*: a value that is taken from the sample where the given percent ($percentile$) of the sample data are under that value.

5. *range*: the difference between the minimum and maximum value of a property

6. *min*: the minimum value from the samples collected about each property

7. *max*: the maximum value from the samples collected about each property

The value of each statistical parameter of each property of each host will be saved for some period of time on a unique variable that is generated by the concatenation of *host-name*, *property* and *statistics* elements. That means if there are two hosts H1 and H2, then $2 \times 11 \times 7 = 154$ variables will be created to save the statistics result (table 5.2).
5.4. PLUGIN PART

Table 5.2: Variables, that are used to store statistics results, are generated by concatenating elements of the three sets.

<table>
<thead>
<tr>
<th>Host-name list</th>
<th>Host-Properties</th>
<th>Statistics parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>(H1, H2, H3 ......)</td>
<td>(count, running, blocked, paused, shutdown, shut-off, crashed, active, inactive, noofcpu, cpuusage, guestmemory)</td>
<td>(mean, median, mode, percentile, range, min, max)</td>
</tr>
</tbody>
</table>

Perl has a variable type called \%hash, which fits for such types of variable deceleration. The following table shows the structure of these variables in Perl:-

```
# variable format : $statistics{$host}{$property}{mean} = result;

$statistics{Host1}{count}{mean} = average-number-of-VMs-on-Host1;
$statistics{Host1}{running}{mean} = average-number-of-running-VMs-Host1;
$statistics{Host1}{cpuusage}{percentile} = percentile-result-of-CPU-usage;
$statistics{HostN}{guestmemory}{max} = the-maximum-memory-used-on-hostN;
```

For example if a user writes rules using a host name 'HostN', then data will be collected from this host, descriptive statistics about each property will be done and stored in a temporary variable that is generated as shown above. If the host name is set by a keyword 'all', then the statistics considers the whole network resource usage.

5.4.1.3 Rules-Block

This part of the skeleton program is processed after each periodical statistics calculation is completed. Originally this is a blank space on the skeleton program. By the time of creating a daemon-plugin, this block will be filled with rule-codes. Rule-codes are generated from each rule in the Rules-file. Each Rule-code has if-condition-action structure that checks whether the given condition matches with the collected statistics values (from statistics-block) and executes the given actions (DFD 5.16). Translating rules into rule-codes is done by the Plugin-process (sec. 5.4.2.2).
5.4. PLUGIN PART

Figure 5.16: Flowchart of *buna* skeleton program that has three separated block-codes named variable, statistics, and rule block.
5.4. PLUGIN PART

5.4.2 Plugin Process

It’s discussed that the purpose of the Plugin-process is to create a daemon-plugin by copying the skeleton program, and doing some modification on the skeleton’s variable-block and rules-block. The steps involved from the beginning of creating a daemon-plugin to executing it are:

- The process starts by creating a new file for the daemon-plugin, the name of the file is taken from –plugin <daemon-name> <rules-file> input.
- Copy the variable-block of the skeleton program to the daemon-plugin file.
- Validate rules file: check each of the rules if hey are written based on buna-rule-syntax or not.
- Update variables: read rules-file to find the host names that are used in each rule and some variables’ values, and take the necessary modification on daemon-plugin.
- copy the statistics-block of skeleton program to the daemon-plugin file.
- Copy the empty rules-block of skeleton program to the daemon-plugin file.
- Read each rule in rules-file, translate to rules-code, and then write it into the rules-block area inside the daemon-plugin.
- Finally run the daemon-plugin as a Linux daemon program.

5.4.2.1 Updating Variables & Validating Rules

As stated in the previous section, this part of the Plugin-Process is used to check whether each rule is written based on buna-rules-syntax or not. If a rule is written incorrectly, it removes that rule from the system. In addition to that if some global variable values are given by the user, it updates them. These values are the host-names that are used in each rule, $samplesize that limits the number of samples going to be collected to do statistics, and $interval that limits the time interval for collecting a sample (Flowchart 5.17).

Once the rule validation and variable updating is completed, it forwards the updated variables to the next step for writing them into the Daemon-Plugin program.
Figure 5.17: Flowchart of plugin-process: the process takes the names of used hosts, modifications of some variables, and checks the syntax of rules written in the rule file.
5.4. PLUGIN PART

5.4.2.2 Generating Rule-Codes

Because rules are written in a simple structure that has a condition and action part, translating each rule to a simple if-condition-action code can be done easily. The condition parts are written by applying mathematical logic on the known host names and keywords; and the action part is any executable program (4.2.2). The translation is done as follows:-

1. Separate the condition and action part of the rule by using ";" character

2. Add prefix and postfix strings on the condition part to get a full if-condition-code, which looks like "if(condition)"; and write it inside rules-block of the skeleton program.

3. Adjust the action part by adding some strings to get the executable structure, which looks like system(action);", and write it next to its opened condition part.

4. Finally close the condition by using "}" at the end.

5. If the one time flag is on, create and write a unique variable that controls the number of times the action will be done.

By repeating the above steps for each rule the full if-condition-action code will be generated and added into the plugin program (Flowchart 5.18).
5.4. PLUGIN PART

Figure 5.18: Flowchart for generating rules-code (if-condition-action code) for each rule of rules-file.
5.4. PLUGIN PART

Samples of rules while they are converted to if-condition-action codes:-

```
Rules written in rules-file

$rule(local.running.mean <= 4; buna --action start --vm vm4);
$rule(local.running.mean > 10; buna --action destroy --vm vm4);

$rule(remote.running.max > 10; buna --connect remote --action destroy --vm net*; 1);
$rule(remote.running.max > 15 or local.running.max > 8; echo overload!!!!);

$rule(remote.running.max > 2; buna --connect remote --action destroy --vm net*);
```

The plugin-process converts the above rules into if-condition-block and writes inside the skeleton program as follows:-

```
#Begin_Rule#
if($statistics{local}{running}{mean} <= 4){
    system( " buna --action start --vm vm4"
}
if($statistics{local}{running}{mean} > 10){
    system( " buna --action destroy --vm vm4"
}

if($statistics{remote}{running}{max} > 10){
    unless($onetime3 eq 'yes'){
        system( " buna --connect remote --action destroy --vm net"
    }
    $onetime3='yes';
}

if($statistics{remote}{running}{max} > 15 or $statistics{local}{running}{max} > 8){
    system( "echo overload!!!!"
}
if($statistics{remote}{running}{max} > 2){
    system( " buna --connect remote --action destroy --vm net"
}
#End_Rule#
```

5.4.2.3 Running Daemon-Plugin as a Linux Process

Once the Daemon-Plugin program is generated, the program will be compiled and executed as a Linux daemon program using a simple command as follows:-

```
# change the Daemon-Plugin file into executable mode:
chmod 755 Daemon-Name

# execute it as a daemon:
./Daemon-Name &
```
5.5 Summary of Buna Design

When a user runs *buna* with the necessary input options, the 'Option-Manager' accepts them and takes the necessary activities like validating input values, changing the default values of global variables, and generating and filtering virtual machine names. Finally it forwards all the ready made data to one of the processes that can accomplish the given task.

There are four major processes ready to do different tasks. A process named *Plugin-Generator* will be called to manage resources between hosts (like power saving and load balancing on CPU and Memory), and to keep VMs status. The other processes named *Action*, *Information* and *Device automations* are used to take actions (like start, stop and create virtual machines); get information (like status of VMs and amount of Memory assigned); and attach/detach devices on virtual machines (DFD 5.19).
Figure 5.19: Context Diagram of *buna*. This image shows the full structure of *buna* interaction with users and libvirt APIs, and data flows between internal processes.
Chapter 6

"buna": Implementation

*buna* can be implemented in many programming languages like C, C++, and Python with some modifications on the design as some of the flow charts contain Perl lines of code. This chapter shows the implementation of *buna* design using Perl programming language, version 5. *Perl* is selected as it is powerful to manipulate strings, easy to work with files, easy to execute system commands and has flexible modules used for system administration purposes [4]. It is also becoming a fully Object Oriented language in its newest version called *rakudo Perl6* [14, 3].

Because *buna* has more than 1,400 effective lines of code, with lots of functions, this chapter explains only the basic code that give a general overview of the whole program. The entire code set is shown in appendix 'E'.

This chapter has two sections: The first section shows implementation of the "automation part" and the second section shows the "plugin part" of *buna*.

### 6.1 Configuration File

The *buna* configuration file is located at `/etc/buna/buna.conf`, this is used to list ssh-users, available-hosts, storage-media, snapshot-storage-directory and XML file storage directory of virtual machines. For better understanding, comments are included inside the code.

#### Listing 6.1: /etc/buna/buna.conf

```
#buna configuration file contains four types of variables
#First-Variable is ---->$USER, a common user for ssh connection to all --hosts. Default is 'root'
#Uncomment and change it if root is not used as a user.
$USER=MethodName;

#Second-Variable is ----> SHOST(alayses-name, uri);
# list all servers name with their uri one by one using the above ----
```
6.2 Rules File

Rules can be written in a file and feed to buna. The default rules file is located in `/etc/buna/buna.rules`; which is used as reference for property and statistics keywords.

Listing 6.2: /etc/buna/buna.rules

```plaintext
# RESOURCE KEY-WORDS with description (read virsh and virt-top manual)

# count: number of existing vms from a host
# active: number working virtual VMs from a host
# inactive: number of power-off VMs from a host
# running: number of running vms form a host

# STATISTICS KEY-WORDS that will be used for each RESOURCES

# mean: the average value of the give sample
# median: the median value of given sample
# mode: the more repeated value of given sample
# percentile: a value where the given percent of samples are under it
# range: the range of sampleset
# min: minimum value from sampleset
# max: maximum value from sampleset

# parameters (can be changed) 

# "samplesize" -> is the number of samples going to be collected about

$SampleSize = 10;
$Interval = 3;
$Percent = 75;

# Example user defined

syntax -> $rule(condition; any action[; flag]);
# flag limits the action execution to be done one time only

$rule(local.running mean <= 4; buna --action start --vm vm4);
$rule(local.running mean > 10; buna --action destroy --vm vm4);
```

6.3 Option-Manager
Listing 6.3: Included Perl Modules & Command Definitions

```perl
#!/usr/bin/perl

use Getopt::Long; # To accept inputs through options
use List::MoreUtils qw{any}; # To refer elements in an array
use List::MoreUtils qw/uniq/; # To remove redundancy in an array
use List::MoreUtils; # To ease string manipulations

# Global Variables For Commands
@EVENT_ACTION = ('clone', 'start', 'destroy', 'suspend', 'resume', 'reboot', 'shutdown', 'save', 'restore', 'migrate', 'undefine');
@UPDATE_ACTION = ('setmem', 'setmaxmem', 'setvcpu');
@STATE = ('running', 'idle', 'paused', 'down', 'off', 'crashed', 'dying', 'all');
@NO = ('dominfo', 'domuuid', 'domid', 'vcpuninfo', 'ttyconsole', 'vncdisplay');
@DEVICE_ACTION = ('attach-device', 'attach-disk', 'attach-interface', 'detach-device', 'detach-disk', 'detach-interface');
@ACT_INACT = ('all', 'active', 'inactive');
```

Groups of commands that will be used as input through --information, --action, --device options are stated (line 11-15). This grouping is based on the parameters needed to process each command. Most commands are taken from virsh manager, one from virt-clone, and some are written by us. Line 16 contains three values of --vm option that are used to represent active, inactive or all virtual machines.

Listing 6.4: Variables’ assigned with default value or from buna.conf

```perl
#!/usr/bin/perl
# Default Values
$USER = 'root'; # common user for ssh connection to all hosts
$HOST = ('local', 'qemu:///system'); # default host
$CONFIG_FILE = '/etc/buna/buna.conf';
$SKELETON_PROGRAM = '/etc/buna/skeleton'; # skeleton for plugin part
$FILE_DIR_LIST[0] = ('/var/lib/libvirt/images/'); # for VMs image file
$DIRECTORY = '/var/lib/libvirt/store/'; # for snapshot
$XML = '/etc/libvirt/qemu/'; # XML file storage directory for VMs

# Reading from configuration file to update default values
open(MYFILE, "$CONFIG_FILE") or die "couldn’t open \"$CONFIG_FILE\" file: $!

while ($line = <MYFILE>) {
    chomp $line;
    $line =~ s/ /;/g;
    $USER = $1 and next if $line =~ /\^s*\$USER\s*=(.*)\s*/;
    $HOST = $1 and next if $line =~ /\^s*\$HOST\s*=(\.(.+\.(.+))\s*)/;
    $FILE_DIR_LIST[$2] = $1 and next if $line =~ /\^s*\$FILE_DIR\s*=(.*)\s*/;
    $DIRECTORY = $1 and next if $line =~ /\^s*\$DIRECTORY\s*=(.*)\s*/;
}
close(MYFILE);
```
Assigning default values for global variables is done (line 1-9). Some of these default values can be changed through buna.conf. Each line of buna.conf will be read and checked against the user input options; then if there is a difference the default values will be changed (10-20).

Listing 6.5: Input Options

```bash
# Handling Flags and Arguments
GetOptions ("help!" => $help, # display buna usage
            "simulate!" => $simulate, # To visualize operations.
            "plugin:s({})" => @plugin, # To run the daemon−plugin
            "vm:s({})" => @step, # To accept VMs−name as a syntax
            "count:i" => $count, # To limit the number of VMs
            "information:s" => $info, # To get VMs information
            "action:s" => $action, # To take actions on VMs
            "live!" => $live, # For live migration
            "connect:s({})" => @connhost, # Target Host
            "destination:s" => $desthost,
            "directory" => $DIRECTORY,
            "remove" => $remove_vms_image,
            "device:s({})" => @device
            );
```

All the above options are declared to interact with users. Options can be expressed using a hyphen, " - ", followed by name of the option. It’s possible to use some part of the option’s name if it is unique from all available options’ name. For example, ‘−con’ is equivalent to ‘−connect’, but ‘−co’ can’t be used as there are more than one options that start with "−co" like '−count' and '−connect'. Options named 'information', 'action' and 'device' handle input values from the user, but these values are limited and defined on listing 6.3.

Listing 6.6: Input Values Validator

```bash
if ($help) {
    usage(); # prints usages of buan
    exit 0;
}

unless (@plugin or $info or $action or @device){
    print "−−help, −−plugin, −−information, −−action or −−device value ←
        is mandatory\n";
    exit 0;
}

if (@connhost) {
    if ($connhost[0] eq 'all') {
        @connhost = keys @HOST;
    }
}
```
The above sample code is a part of input value validator. When a user runs `buna`, whether the basic input options are used or not will be checked (11-14). Presence or absence of `uri` records for host names, given through `-connect` option, is also checked (14-22). If no `uri` is given, 'local' will be taken as a default `uri` (line 23).

### 6.4 Name-Generator and Filter

Listing 6.7: Separating the cross product areas and also their sets

```perl
sub vm_syntax_filter {
    my @vms_all; # storage for full names
    my @cross = split('}', $_[0]); # separate the cross product area
    for each $cross (@cross) {
        next if ($cross eq "");
        my @vms_cross = 1;
        # separate sets in a cross product range
        my @sets = split(';', $cross);
        for each $set (@sets) {
            chomp $set;
            next if ($set eq "");
        }
    }
}
```

Full names of virtual machines will be stored in a variable (line 3). First the `vm-name-syntax` given through `-vm` option will be taken and split into strings that represent each cross product area (line 4); then sub-strings that represent each set of cross product areas will be populated by using `'*'` (line 5). The next step is expanding sequential numbers.

Listing 6.8: Expanding sequential numberings

```perl
while (1) {
    if ($set =~ /^([\^]*)(\d+)(\d+)?(\d+)?([\^]*)(\s*)\)/) {
```
6.4. NAME-GENERATOR AND FILTER

my $pre = $1;
my $min = $2;
my $max = $4;
my $post = $5;

# if ($pre =~ / (.*) (\d+) / ) {
  if ($min >= $max) {
    my $t = $min;
    $min = $max;
    $max = $t;
  }
  my $text;
  foreach my $i ($min .. $max) {
    $text =~ s / (\^[^\s]*\$)\d+(?: (\.)\d+(?: (\^[^\s]*\$))* ) / ;
  }
  $set =~ s / (\^[^\s]*\$)\d+(?: (\.)\d+(?: (\^[^\s]*\$))* ) / ;
  } else {
    last;
  }
}

For each set of string, the presence of sequential numbers is checked and if exist, it identifies the prefix, postfix and the minimum and maximum numbers (2-15). Then name population will be done by concatenating the prefix, integer($i) and postfix; where $i is a number between the minimum and maximum numbers (15-25). Finally these expression part will be replaced with the expanded string (line 20) and searching the next one will continue (line 1 and 25). The next step will be generating the full names of virtual machines.

Listing 6.9: Generating full names of virtual machines

    @set_expand = split('\s', $set); # separate elements of sets
    @temp = @vms_cross; # put the root part in temporary store
    @vms_cross = (); # cross product between sets to get full name
    foreach my $temp (@temp) {
        if ($Temp) {
            if ($Temp =~ /\s/) { # stop further cross products ← through it
                push @vms_cross, $temp;
            } else {
                foreach my $set_expand (@set_expand) {
                    if ($set_expand) {
                        my @string = $temp . $set_expand;
                        push @vms_cross, @string;
                    }
                }
            }
        } else {
            foreach my $set_expand (@set_expand) {
                if ($set_expand) {
                    my @string = $temp . $set_expand;
                    push @vms_cross, @string;
                }
            }
        }
    }
Elements of a set will be separated from the sub-string using white-space as delimiter (Line 4). Concatenating elements of the current set to the root part is done (line 10-16); but terminated elements of the root part, those with a period "." at the end, will not be appended (line 10). This procedure continues until all sets in the cross product area are finished. The last line shows returning of the full name generated in that cross product area.

6.5 Type-Checker

Listing 6.10: Calls the right process going to continue

Based on the user input, it calls the function to process the appropriate task. For example, line 2 calls the clone process if the user inputs '–action clone' to buna. Line 4 calls the update process if the input value is any one from the specified array. All if..else conditions are referring already defined values stated on listing 6.3.

6.6 Automation Part

6.6.1 Information Process

Listing 6.11: List Virtual Machines using the given status

# Gathering the status of VMs like running, idle, crashed, ...., and all

if (any { $_.eq $info} @STATE) {
  print "from: $connhost($CONNECT_URI)\n";
  if ($info eq 'all'){
    $command = 'virsh --connect '$CONNECT_URI' list --all' ;
  open(MYFILE, "$command |") ;
  @lines = <$MYFILE> ;
  splice @lines, 0, 2 ;
  } else {
    $command = 'virsh --connect '$CONNECT_URI' list --all | grep "$info" ;
  open(MYFILE, "$command") ;
  @lines = <$MYFILE> ;
  splice @lines, 0, 2 ;
  }
6.6. AUTOMATION PART

```perl
open(MYFILE, "$command") or die;
@lines = <MYFILE>;
}
# Gather information for the given VMs
if ($step)
  my @templines;
  @vm = vm_syntax_filter($step); # call VMs name generator
  foreach $line (@lines)
    foreach $vm (@vm)
      if ($line =~ /\$vm/)
        push @templines, $line;
  }
  @lines = uniq @lines;
  @lines = @lines[0 .. $count - 1] if ($count > 0 and $count < @lines);
print "@lines
```

Line 2 checks whether the input value exists in the @STATE (listing 6.3) through –information option. If that value is ‘all’ (means the user needs all virtual machines in any state), then the ‘virsh list –all’ command will be executed on the targeted hypervisor and unwanted dotes and spaces will be removed (line 9). Then it’s returned to the user (line 30). But, if the request is getting the status of specific virtual machines (16), full names of virtual machines will be generated (line 18); all virtual machines with their status will be collected (line 11-13) and filtered (19-27). Finally the filtered virtual machines will be returned to the user, including their status (line 30).

6.6.2 Action Process

6.6.2.1 Clone-Action

Listing 6.12: Filtering Template Virtual Machines from the given name-syntax

```perl
while (1)
  if ($templateString =~ /^([a-zA-Z_0-9-]+)\s*:\s*([a-zA-Z_0-9-]+)\s*$/)
    $group=$1;
    $template=$2;
    chomp $group;
    chomp $template;
    $templateVM{$group}[0]=$template;
    chomp $2;
    $command = "virsh --connect $CONNECT_URI dumpxml -v $templateVM | grep '<source file=' | awk -F'\n' '{print \$2}';";
    $file_path = $command ;
```

99
Before full names of virtual machines are generated from the given vm-name-syntax, the template virtual machines will be filtered out from the syntax. For this, the vm-name-syntax ($templateString) will be checked if it has a template VM($template). The virtual machines group together with the template virtual machine name will be stored in a hash variable with key and value pairs (3-7). The file path of the template will be checked (8-17) and its file size will be stored inside the hash variable as a second element (19-35). The analyzed templates will be removed from the string (36) and the process continues until all templates are cross-checked (36-40). Doing these steps helps to give different templates for groups of virtual machines, even if they are in the same vm-name-syntax. Next step will be checking the available free storage size from the host.

Listing 6.13: Checking the available storage space

```
$command= "full_user_name" df | awk '{ print "$6 \"\" "$4 }\"" ";
@free_space= $command ;
foreach $line (@free_space){
  ($path, $freeKB) = split ("\", $line);
  chomp $path;
  chomp $freeKB;
```
The available mounted storage space is collected using ‘df’ Linux command (1-8). Different mounts might represent a file system partition and to remove such redundancy, checking their original partition is done to select a single mount place per partition (9-35). Once this is done, cross checking of the required storage space for the new virtual machine will be done.

Listing 6.14: List Virtual Machines going to be created together with the proposed storage path.

```bash
@vms=vm_syntax_filter($step); # call VMs name generator
my $size=0; # increment of storage size of each VM
my $count=0; # counting VMs
my @groups=( keys %templateVM );
my $group=join(' | ', @groups );
my @groupedVMs = grep /,$group/, @vms; # filter VMs that are in a ← template group.
@vms= @groupedVMs;

foreach $path (keys %filter_dir) {
    $size=0;
    $full=0;
    ```
6.6. AUTOMATION PART

The variable %templateVM contains template-virtual-machines corresponding to the assigned group-name as key and value pair. Where group-names are set as keys, and template-names and their-size as values of the hash variable (listing 6.12). After collecting the generated virtual machines’ names (line 1), names which don’t have assigned template VMs will be rejected (4-6). Then the available storage media will be checked to see how much virtual machines they can support (9-25). Information will be displayed to the user (29-33) and also stored in the variable $stack (line 31). The user will be warned if there is no enough space for some virtual machines (34-38). If the user agrees to continue cloning will be done using the collected information.

### 6.6.2.2 Rename-Action

Listing 6.15: buna code for renaming virtual machines
At the beginning, the new virtual-machine-name is formulated (9-11), the XML file is dumped into an array (13, 14), then the new XML and file-path of the virtual machine is generated (18-21). Each line of the array is written in a temporary file: through this process, the virtual-machine-name and image-file path will be modified with the new name (23-32). After the XMI modification is finished, the name of image file is changed to the new virtual machine name (33, 34), the temporary XML file is saved with the new name (35, 36), and the old XMI file is removed from the host (37, 38). After doing the above steps for each virtual machine (8-39),
6.6. AUTOMATION PART

libvirt daemon will be restarted (40-44).

6.6.2.3 Event-Action

Listing 6.16: Migrating Virtual Machines from one Host to another

```perl
sub Sub_Migrate{
    print "−−vm is mandatory\n" and exit 0 unless (@step);
    if ($SOURCE_URI eq $DESTINATION_URI) {
        print "migration with same −−source and −−destination not ←
        possible\n";
        exit 0;
    }
    @vm = vm_syntax_filter($step); # call VMs name generator
    @vm = @vm[0 . . $count - 1] if ($count > 0 and $count < @vm);
    foreach my $vm (@vm){
        if ($live){
            $command = "virsh migrate −−live " . $vm . "$DESTINATION_URI ←
            $SOURCE_URI";
        } else {
            $command = "virsh migrate −−suspend " . $vm . "$DESTINATION_URI ←
            $SOURCE_URI";
        }
        if ($simulate){
            print "$command\n";
        } else {
            system("$command");
        }
    }
    exit 0;
}
```

First the source and destination host uri will be checked (3-6), and virtual machine names will be collected (7), then virsh migrate command will be formulated and executed for each virtual machine (7-22). The input flag ‘−−live’, will also be checked to see if the user needs live migration (10-12).

Listing 6.17: Undefine Virtual Machines. removing their images of necessary

```perl
sub Sub_Undefine{
    # adjust ssh connections
    @vm = vm_syntax_filter($step); # call VMs name generator
    @vm = @vm[0 . . $count - 1] if ($count > 0 and $count < @vm);
    chomp($action);
    # undefine VMs
    foreach my $vm (@vm){
        chomp($vm);
        if ($remove_vms_image){
            print "removing image file ....\n";
            $command = "virsh −−connect $CONNECT_URI dumpxml $vm | ←
grep 'source file=' | awk -F\"' \{ print \$2\}";
        }
        # undefine VM $vm
    }
}
```
6.6. AUTOMATION PART

```perl
$file_path = $command;
chomp $file_path;
print ":\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\\n"
;system("$full_user_name rm -r $file_path");
}

if ($vm) {
    print "\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\\n"
;system("virsh --connect "$CONNECT_URI undefine $vm"");
}

if ($simulate) {
    print "$command
";
} else {
    system("$command");
}
```

The above code removes virtual machines from the host. For each virtual machine, `virsh undefine` command will be formulated and executed (20-30). Virsh can’t remove the image files of virtual machines when it removes the VMs. To do so, `buna` adds an additional input option `–remove`. When these flag is added, `buna` removes the image of the virtual machines before they are undefined (11-18).

Listing 6.18: Other event actions run with the same function

```perl
sub Sub_Event_Action {
    # adjust ssh connection
    @vm = vm_syntax_filter($step);  # call VMs name generator
    @vm = @vm[0 .. $count - 1] if ($count > 0 and $count < @vm);
    chomp ($action);  # generate command and take action
    foreach my $vm (@vm) {
        chomp($vm);
        if ($vm) {
            print "\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\\n"
;system("virsh --connect "$CONNECT_URI $action $vm"");
        }
    $vm = "";
}
```

105
6.6. AUTOMATION PART

All other actions in @EVENT_ACTION, like `pause`, `resume` and `stop`, will be processed together with the above code. The users input value of `-action` option will be validated (5); and for each virtual machine name the virsh command will be formulated and executed (9-22).

6.6.2.4 Update-Action

Listing 6.19: Changing virtual machine names

```perl
foreach my $vm (@vms) {
    $new_vm_name = $vm;
    $new_vm_name =~ s/$name[0]$/name[1]/;
    next if ($vm eq $new_vm_name);
    $command = 'virsh --connect SCONNECT_URI dumpxml $vm | grep -w <source file= | awk -F \" { print \$2 } " ;

    $old_file_path = `$command` ;
    if ($old_file_path =~ / (.*) / (.* ) (\.*) /g ) {
        $new_file_path = $1 . $new_vm_name . $3 ;
        $old_xml_path = XML.$vm . ' . xml ' ;
        $new_xml_path = XML.$new_vm_name . ' . xml ' ;
        open (OLDXML, "< $old_xml_path") or die "Unable to open" ;
        open (NEWXML, "> $new_xml_path") or die "Unable to open" ;
        while (my $var = <OLDXML>) {
            if ($var =~ / <name>$vm</name> / ) {
                $var = " <name>$new_vm_name</name> \n " ;
            } elsif ($var =~ /<source file=.*>/) {
                $var = "<source file="$new_file_path"/>\n" ;
            }
            print NEWXML "$var" ;
        }
        close (OLDXML) ;
        close (NEWXML) ;
    }
    $command = "$full_user_name my $old_file_path \n $new_file_path" ;
    system ( "$command" ) ;
};
```  

The above code modifies the name of virtual machines that is used for regrouping VMs group. This code will be invoked when buna runs as `buna --action chname --vm vm-name-syntax --change old-string new-string`. From each virtual machine name the new name will be generated by replacing the given `old-string` with the `new-string` (2-6). The image storage path of the virtual machine will be handled from its old XML file (7). The Old XML file of the virtual machine will be moved to the a new XML file with the new name(13-24), while moving it the VMs `name` and `image-file-storage-path` will be modified (16-20). Finally the VM image file is changed from its old name to the New name (25-26).
Listing 6.20: Changing the number of CPUs and Memory size of VMs

```perl
sub Sub_Update_Action{
    print "from: $hostname($CONNECT_URI)\n";
    @vms = vm_syntax_filter($step); # call VMs name generator
    @vms = @vms[0 .. $count - 1] if ($count > 0 and $count < @vms);
    # updating action
    foreach my $vm (@vms) {
        my $command = 'virsh --connect ' . $CONNECT_URI . $action . $vm . $count;
        if ($simulate) {
            print "$command\n";
        } else {
            system("$command");
        }
    }
}
```

The above simple code will be executed when the input value of `action` is in the list of `@UPDATE_ACTION=('setmem', 'setmaxmem', 'setvcpus')`(listing 6.10). For each virtual machine, the virsh update command will be formulated and executed (9-19).

### 6.6.3 Device Process

Listing 6.21: Attaching and Detaching devices to Virtual Machines

```perl
sub Sub_Device_Action{
    if (any {$_ eq "$device[0]"} @DEVICE_ACTION) {
        @vms = vm_syntax_filter($step); # call VMs name generator
        @vms = @vms[0 .. $count - 1] if ($count > 0 and $count < @vms);
        splice @device, 1, 0, $vms;
        # generate virsh command and take device action
        foreach my $vm (@vms) {
            chomp($vm);
            splice @device, 1, 1, $vms;
            # splice @device, 1, 0, $vm;
            $device="@device";
            $command = 'virsh --connect ' . $CONNECT_URI . $device . $device;
            if ($simulate) {
                print "$command\n";
            } else {
                system("$command");
            }
        }
    } else {....
```
6.7. PLUGIN PART

Attaching/Detaching device is a little bit different from information and action virsh commands. virsh manager uses lots of different options to attach/detach a disk, interface or other device. A better approach that buna follows is to allow the users insert all options as they are used in virsh manager, but without the virtual machine’s name. When the device is going to be attached/detached, each virtual machine name will be inserted at the second place from the input value (line 12) and virsh command will be generated and executed (14-21).

6.7 Plugin Part

6.7.1 Skeleton program/ Daemon-Plugin

From the beginning, used Perl modules for the daemon are included (1-8). A variable that contains list of host properties is defined (line 10), where data will be collected about these properties from each host and statistics will be done. Rules in rules-file also use these keywords to refer the statistics data. The free space shown between line 14 and 18 will be filled with variables collected from Rules-File (listing 6.2). When the new Daemon-Plugin is generated by the Plugin-Proces, the free space will be filled as follows:-

, while the new Daemon-Plugin is generated suing the skeleton & Rules-File.

Listing 6.22: Used Perl modules & Global Variables

```
#!/usr/bin/perl

# Used packages
use Getopt::Long;
use List::MoreUtils qw{ any }
use List::MoreUtils qw/ uniq /;
use List::MoreUtils;
use Statistics::Descriptive;

# Global Variables
@head = ('count', 'running', 'blocked', 'paused', 'shutdown', 'shutoff', 'crashed', 'active', 'inactive', 'noofcpu', 'cpusage', 'physicalmemory', 'guestmemory');

# Begin Variable #
# variables for hosts, SampleSize, and Timelimit will be stored here:

# End Variable#
```

Listing 6.23: Variables Collected from Rules-File or filled with default values by plugion-process

```
#Begin_Variable#
$SampleSize = 40;
```
6.7. PLUGIN PART

\$Interval = 60;
\$Percent = 80;
$hostlist=('local', 'qemu:///system', 'remote', 'qemu+ssh://192.168.0.27/→
system');

#End_Variable#

Listing 6.24: Collecting sample data from hosts using virt-tom manager

```perl
foreach $host (keys %hostlist) {
    $pid = fork();
    if ($pid) {
        # parent
        $childs[$count]=$pid;
    } elsif ($pid == 0) {
        # child areas
        $datafile="virttop" . "$host";
        $command = "virt-top --connect '/$hostlist' --n ← $SampleSize ←d $Interval ←s $script ←csv $datafile";
        system("$command");
        exit 0;
    } else {
        die "couldn't fork: $!
    }
    $count++;
}
```

For each host a child process will be created using fork function (1-6). The child process will execute the virt-top command using the given sample-size and time-interval. Then data collection will start at the same time on all hosts (7-16).

Listing 6.25: Filtering Property data and call for statistics

```perl
foreach $host (keys %hostlist) {
    $datafile="virttop" . "$host";
    open(MYFILE, "$datafile") or die "couldn't open the file \n";
    @lines = <MYFILE>;
    close(MYFILE);
    system('killall -9 virt-top');
    splice @lines, 0, 2;
    $i=0;
    foreach $line (@lines) {
        @columens = split(' ', $line);
        $data{$host}{noofcpu}[$i] = $columens[3];
        $data{$host}{running}[$i] = $columens[5];
        $data{$host}{paused}[$i] = $columens[7];
        ...
        $data{$host}{guestmemory}[$i] = $columens[16];
        $i++;
    }
    foreach $type (@head) {
        @statf=analyze(@$data{$host}{$type});
    }
```

109
6.7. PLUGIN PART

```bash
$statistics{host}{type}{mean}=$statf[0];
$statistics{host}{type}{median}=$statf[1];
$statistics{host}{type}{mode}=$statf[2];
$statistics{host}{type}{percentile}=$statf[3];
$statistics{host}{type}{range}=$statf[4];
$statistics{host}{type}{min}=$statf[5];
$statistics{host}{type}{max}=$statf[6];
```

The samples of each property of each host will be filtered from the collected data; `virt-top` stores these samples in a line separated by comma (1-18). Then, using these samples, descriptive-statistics will be done for each property of each host (20-30). This will be done by calling the `statistics` function (listing 6.27). Initially, the Rules-Block of the skeleton program is blank space (31-35). When a Daemon-Plugin is generated, this space will be filled with if-condition-action codes; these are translated from rules as shown in the example below.

**Listing 6.26: Condition-Block of a Daemon-Plugin**

```bash
#Begin_Rule#
if($statistics{local}{running}{mean} < 4){
system("buna --action start --vm vm4");
}

if($statistics{remote}{running}{max} < 5){
unless($onetime3 eq 'yes'){
system("echo overload!!!");
system("buna --connect remote --action start --vm net*");
$onetime3='yes';
}
}
#End_Rule#
```

The above two if-condition-action codes are generated from two rules of a Rules-File. The first condition checks if the number of running virtual machines in the local host is less than 4 or not: If it’s less than 4, the virtual machine named 'vm4' will be started using `buna` action command. The second condition checks if the number of running virtual machines is less than 5; then all virtual machines whose name begins with 'net' will be started; an "overload!!!" message will also be displayed. The one time variables are used to take action only once (line 8 and 11).
6.7. PLUGIN PART

Listing 6.27: Descriptive Statistics Calculator

```perl
sub analyze {  
    my @array = @_;  
    my $stat = Statistics::Descriptive::Full->new();  
    $stat->add_data(@array);  
    my $mean = sprintf("%.2f", $stat->mean());  
    my $median = sprintf("%.2f", $stat->median());  
    my $mode = sprintf("%.2f", $stat->mode());  
    my $percentile = sprintf("%.2f", $stat->percentile($Percent));  
    my $min = sprintf("%.2f", $stat->min());  
    my $max = sprintf("%.2f", $stat->max());  
    my $range = sprintf("%.2f", $stat->sample_range());  
    return ($mean, $median, $mode, $percentile, $range, $min, $max);  
}
```

This function accepts samples of data as an array and does descriptive statistics. Line 3 contains a CPAN module used for descriptive statistics calculation. The result will be used for if-condition-action evaluation part.

6.7.2 Plugin-Process

Listing 6.28: Listing keywords of host properties & descriptive statistics

```perl
sub Sub_Plugin {  
    @property = ('count', 'running', 'blocked', 'paused', 'shutdown', 'shutoff', 'crashed', 'active', 'inactive', 'noofcpu', 'cpuusage', 'physicalmemory', 'guestmemory');  
    @stat = ('mean', 'median', 'mode', 'percentile', 'range', 'min', 'max');  
    ($PLUGIN_NAME, $RULE_FILE) = (@plugin);  
    $RULE_FILE = '/etc/buna/buna.rules' unless ($RULE_FILE);  
    $PLUGIN_NAME =~ s/^[a-zA-Z_0-9-]*//g;  
    # assigning default values  
    $SampleSize = "40";  
    $Interval = "60";  
    $Percent = "80";  
}
```

Lines 2 and 3 define the available host properties and descriptive statistics. Line 4 takes input values (Daemon-Plugin-Name and Rules-File) from –plugin option. Lines 8-10 represent default values of some variables that will be modified through Rules-File.

Listing 6.29: Validating Rules-File and updating values of global variables

```perl
$LineCount = 1;  
# Validating rules, filtering names of used hosts and defined variables' values  
my %RuleHost;  
my @CancelledRules;  
open(MYFILE, "$RULE_FILE") or die "couldn't open \"$RULE_FILE\" file\n\n\n";  
while ($line = <MYFILE>){
```
6.7. PLUGIN PART

```perl
chomp $line;

# reading variable value to update the default

$SampleSize = "$1;" if ( $line =~ /[\^\s]*$SampleSize\s*=\s*([^d+]+)\s*;/ ) ;

$Interval = "$1;" if ( $line =~ /[\^\s]*$Interval\s*=\s*([^d+]+)\s*;/ ) ;

$Percent = "$1;" if ( $line =~ /[\^\s]*$Percent\s*=\s*([^d+]+)\s*;/ ) ;

# check the syntax of rules (host, property and statistics -> keywords)

if ( $line =~ /[\^\s]*$rule\(([^s(]+)\s*)/ ) { my @rule = split ( '";"', $line ) ; while ( $rule[0] =~ /[\^\w+]+/ ) { if ( any { $._ eq $2 } @property and any { $._ eq $3 } @stat ) { $RuleHost{ $1 } = $HOS { $1 } ; } else { print "WARNING!!! property/statistics problem -> $2, $3
" ; print " $line
" ; push @CancelledRules, $line ; } } else { print "WARNING!!! host problem->$1, see the following->rule
" ; print " $line
" ; push @CancelledRules, $line ; } } $rule[0] =~ s /[\^\w+]+/ / ; } $LineCount++ ;

close (MYFILE) ;
```

Line 3 defines a hash variable that stores host-names and their uri, which are used in Rules-File. Line 4 defines a variable that is used to store rules that don’t follow buna-rules-syntax. The Rules-File will be opened (5-6), and the default value of some global variables will be changed, if there is a change written in Rules-File (9-11). Each rule will be evaluated whether or not they are written using the available host-names and keywords (14-37). Rules that have problems will be exempted from further process (23-28); also host-names will be collected from each rule and added to $RulesHost variable (22), where these hosts will be considered in this daemon-process. Next step is creating the daemon-plugin using skeleton program and the collected information.

Listing 6.30: Writing the New Daemon-Plugin Program. Adding Variables

```bash
# Writing the new plugin script
```

112
Create a new file using the given daemon-plugin-name, and copy the top part of the skeleton program into this file (2-5). Write the updated variables that are collected before(listing 6.29) into the specified place (7-12). The next step will be converting rules into rule-code and write them to this file.

Listing 6.31: Translating Rules into Rules code and write to the Daemon-Plugin

```perl
if ($SKELETON_PROGRAM =~ /\#Begin_Rule\#/ ) {  
    $LineCount=0;  
    $rulecount=0;  
    # Changing each rule to a code  
    open (MYFILE, "RULE_FILE") or die "couldn’t open ‘$RULE_FILE’ ← file: $!

    while ($line =< MYFILE) {  
        chomp $line;  
        if (any {$_ eq $line} @CancelledRules) {  
            next;  
        }  
        @onetime;  
        if ($line =~ /\s*\$rule\(\s*\)/) {  
            $rulecount++;  
            @rule = split (',. ', $line);  
            $rule[0] =~ s/\((\w+)\:(\w+)\:(\w+)/$statistics\{$1←
            \}{$2\}{$3\}/g;  
            print PLUGIN "if ($rule[0]) \n"  
            @order = split (',', $rule[1]);  
            print PLUGIN "unless ($onetime$rulecount eq ‘yes’) \n←
            if ($rule[2]);  
            foreach $order (@order) {  
                $command = 'system(‘. ‘$order ‘) ‘ if ($order);  
                print PLUGIN "$command" ‘\n";  
            }  
            print PLUGIN "$onetime$rulecount=’yes ‘\n" if ($rule[2]);  
            print PLUGIN "$\n" if ($rule[2]);
```
6.7. PLUGIN PART

Incorrectly typed rules are removed (7-13) and correctly typed rules condition and action part will be separated (16-19). The condition part will be formulated as Perl if-condition and printed into the daemon-plugin-file (20). Each action of the action part of the rule will be identified using comma as a delimiter (21) and printed into the file in Perl executable form (24-27). If the One-time-flag is on, a new variable to this rule will be generated and printed next to the if-condition part (23). Then print the closing bracket symbol at the end to close the opened if-condition (30). The final step is executing this file as a Linux daemon by the following steps.

Listing 6.32: Running the Daemon-Plugin as a Linux Process. It is done by changing the daemon-plugin-file into executable form and run as a background process.

```perl
# Start running the new plugin
system("chmod 777 $PLUGIN_NAME");
system("perl $PLUGIN_NAME &");
exit 0;
```

```perl
print PLUGIN " }\n"

$LineCount++;

} } close (MYFILE);

} 

} close (PLUGIN);
```
Chapter 7
"buna": Testing & Performance Evaluation

The previous chapter has shown the complete implementation of buna using Perl programming language. In this chapter a buna testbed is created based on a university scenario and then the performance of buna is tested for the following characteristics:

1. **Capability**: Checking the availability of buna options for managing the required services of virtual machines.

2. **Simplicity**: To evaluate how simple buna is to install and configure. How easy it is to understand and to be used for accomplishing a task. For example, to start a group of virtual machines, what is needed, which options have to be used and how to use them will be the validation of its simplicity.

3. **Flexibility**: This is to evaluate how buna can be multi-operational to take actions on virtual machines. For example the capability of accessing virtual machines in different groups.

4. **Powerfulness**: How buna can handle the creation and managing of thousands of virtual machines based on organizational needs.

To accomplish this experiment a naming plan of a university scenario is done on the following section.

### 7.1 Naming Plan for a University Scenario

For the university scenario stated in section 4.1, assume system administrators plan to deploy virtual machines for some departments in the following way (see table summary 7.1).
7.2. BUNA TESTBED

Table 7.1: Sample plan for the university scenario.

<table>
<thead>
<tr>
<th>Department</th>
<th>No. of students</th>
<th>Operating System</th>
<th>Template VM</th>
</tr>
</thead>
<tbody>
<tr>
<td>NSA</td>
<td>1st year – 10</td>
<td>both Ubuntu &amp; Windows2010 server</td>
<td>UbuS10-Temp</td>
</tr>
<tr>
<td></td>
<td>2nd year – 6</td>
<td>Windows2010 server</td>
<td>WinS10-Temp</td>
</tr>
<tr>
<td>Biology</td>
<td>3rd year – 12</td>
<td>Windows-7</td>
<td>Win7-Temp</td>
</tr>
<tr>
<td>Chemistry</td>
<td>3rd year – 12</td>
<td>Windows-7</td>
<td>Win7-Temp</td>
</tr>
<tr>
<td>Architecture</td>
<td>5th year – 8</td>
<td>Ubuntu-Desktop</td>
<td>UbuD-Temp</td>
</tr>
</tbody>
</table>

1. The Network and System Administration (NSA) department has 10 first-year and 6 second-year students. Each student needs two virtual machines, one is installed with *Ubuntu-server* and the other is installed with *Windows-2010 server*. To deploy these virtual machines the administrators decide to have two template virtual machines named *UbuS10-Temp* for Ubuntu server VMs and *WinS10-Temp* for windows server virtual machines. The names of the virtual machines are planned to be in the form of ‘UbuS10-NSA-1st-vm1’ and ‘WinS10-NSA-1st-vm1’.

2. The Biology and Chemistry departments each have 12 third-year students. These students require one virtual machine installed with *Windows-7*. Assume administrators plan to create a template virtual machine called *Win7-Temp* and duplicate it for these 24 students at a time. The names of the virtual machines are planned to be in the form of ‘Win7-Bio-3rd-vm1’.

3. The Architecture department has 8 fifth-year students, where these students need *Ubuntu-Desktop* virtual machines. Assume administrators plan to create one template called *UbuD-Temp* that is installed and configured with Ubuntu-desktop, and duplicate it for all students. The names of the virtual machines are planned to be in the form of ‘UbuD-Arc-5th-vm1’.

7.2  buna Testbed

To implement the above plan, the testbed (Fig. 7.1) of *buna* is prepared using the following steps:

1. Three hosts named H1, H2 and H3 have been connected to each other. H1 will be used for NSA, H2 is for both Biology and Chemistry, and H3 is for Architecture department students. Each of these hosts has two Intel(R) Core(TM)2 Duo CPU E7500 @ 2.93GHz, and two 250GB SCSI disks of model ST9250410AS.
2. For this testbed, assume that all hosts are installed with Ubuntu-server 10.04 and libvirt. In reality however, any Linux platform can be selected as long as libvirt can be configured on it.

3. KVM/QEMU hypervisor is installed on all Hosts. Any combination of hypervisors can be used as long as their virtual machines can be managed using libvirt managers.

4. The libvirt management tools: `virt-manager`, `virsh`, `virt-install` and `virt-clone` are installed and configured on all H1, H2 and H3. These libvirt managers must be checked to see if they are functioning properly; because `buna` automates these tools at the background.

5. Create the template virtual machines. This task is not the responsibility of `buna`, rather administrators have to do it using libvirt managing tools (eg. `virt-manager` and `ubuntu-vm-builder`). But the template virtual machines are going to be created based on the university’s plan (sec. 7.1). Because of the resource and time limitations, all template virtual machines are made from same type of operating system and then changed their names with the given template virtual machine names. There shall not be any impact shown on the experiment, because `buna` and the used `libvirt-managers` are not dependant on the type of operating system running on virtual machines. So that ubuntu-hardy virtual machines with default options (a user-name ubuntu, password ubuntu) are created on H1, H2 and H3, and named as ‘UbuS10-Temp’, ‘WinS10-Temp’, ‘Win7-Temp’ and ‘UbuD-Temp’.

```
root@H1:~# ubuntu-vm-builder kvm hardy --name UbuS10-Temp --libvirt qemu:///system
root@H2:~# ubuntu-vm-builder kvm hardy --name Win7-Temp --libvirt qemu:///system
root@H3:~# ubuntu-vm-builder kvm hardy --name UbuD-Temp --libvirt qemu:///system
```

6. `buna` will be installed and configured on H1, in such a way that it can manage virtual machines and hosts’ resources of H1, H2 and H3. For the time being `buna` doesn’t have a ready-made installation package (this can of course be done on other versions). But to make it work now `buna` can be configured as follows:

(a) Install Perl5 and CPAN modules `Getopt::Long` and `List::MoreUtils` on H1 as `buna` depends on them.

```
apt-get install perl

```
7.2. BUNA TESTBED

(b) Create a directory called '/etc/buna/' and copy the 'skeleton', 'buna.conf' and 'buna.rules' into this directory. Copy the main program- buna into '/bin/' directory.

(c) Configure /etc/buna/buna.conf in the following way.

```
$HOST=(local,qemu:///system);
$HOST = (h2, qemu+ssh://192.168.0.27/system);
$HOST = (h3, qemu+ssh://192.168.0.47/system);

$FILE_DIR=/var/lib/libvirt/images/iscsi,0;
$FILE_DIR=/var/lib/libvirt/images/iscsi2,1;
$FILE_DIR=/var/lib/libvirt/images/iscsi3,2;

. default is taken for others
```

From the above, all hosts are listed with their host-name and uri; where local is a synonym for H1. The last two iSCSI type storage files are set to be used for common storage media. These storages will be used to store images of virtual machines that are going to be migrated between hosts. For example, if there are VMs’ to be migrated between H1 and H2, their images must be stored in iscsi2 (sec. 3.9.1).

7. Finally, use buna to deploy virtual machines based on the university plan (table 7.1), and use it for managing the whole network.

The figure below shows the complete structure of buna experiment setup and its implementation.
7.2. BUNA TESTBED

Figure 7.1: Testbed for buna experiment: both hosts are installed with ubuntu-server-10.04 and libvirt. H1 is configured with buna, and also the template VMs to NSA department are hosted here. Similarly H2 is ready for Biology and Chemistry VMs, and H3 is for Architecture department VMs.

The actual implementation of the testbed (Fig. 7.1) is shown below. This is checked by running buna to list all available VMs on all hosts. From the result below the host name 'local' is the synonym for 'H1'.

```
root@h1: # buna --inf all --vm all --con all.
```

```
From: H2(qemu+ssh://192.168.0.27/system)
- Win7-Temp shut off

From: local(qemu:///system)
- UbuS10-Temp shut off
- WinS10-Temp shut off

From: H3(qemu+ssh://192.168.0.47/system)
- UbuD-Temp shut off
```
7.3. BUNA TESTING

7.3 buna Testing

In this section the actual operation of buna is evaluated on the testbed.

7.3.1 buna Usage

The following table shows all options of the first version of buna:

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>root@h1: # buna -h</td>
<td></td>
</tr>
<tr>
<td>buna [-connect &lt;Host-list&gt;]</td>
<td></td>
</tr>
<tr>
<td>--information</td>
<td>action</td>
</tr>
<tr>
<td>--vm &lt;VM-name-list&gt;</td>
<td></td>
</tr>
<tr>
<td>--directory &lt;full-path&gt;</td>
<td></td>
</tr>
<tr>
<td>--count [memory-size</td>
<td>number-of-cpu</td>
</tr>
<tr>
<td>--change &lt;old-substring-of-VMs-name&gt;</td>
<td>new-string-for-VMs-name&gt;</td>
</tr>
<tr>
<td>--xml &lt;directory-of-VMs-xml-file&gt;</td>
<td></td>
</tr>
<tr>
<td>--&lt;live</td>
<td>remove</td>
</tr>
</tbody>
</table>

10   11 12
13   14
15   16
17   18
19   20
21   22
23
24   25
26   27
28   29
30
31   32
33
34
35
36
37
38
39
40
41
42

to print this usage : buna --help

--simulate(-s) : is used to simulate actions

to run the plugin:
  buna --plugin <PLUGIN-NAME RULE-FILE>

to get information:
  buna --information <INFO-VALUES> ...
       [-connect <LIST-OF-HOSTS>] ...
       --vm <VMs-name> ...
       [--count <number-of-VMs | memory-size-in-KB | number-of-CPUs>]

  to take action:
  buna --action <ACTION-VALUES> ...
       [-connect <LIST-OF-HOSTS>] ...
       --vm <VMs-name> ...

  to attach/detach device:
  buna --device <DEVICE-VALUES> ...
       [-connect <LIST-OF-HOSTS>] ...
       --vm <VMs-name>

VALUES:

--plugin <PLUGIN-NAME RULE-FILE>

PLUGIN-NAME : the name of the plugin script when it runs as a daemon

As shown above the capability of buna is broad as far as virsh and virt-clone managers are doing. In addition to that it has different capabilities like: changing VM names and regroup them (line 7, 8), controlling resource of hosts (line 38).
7.3. BUNA TESTING

7.3.2 Automation Test

7.3.2.1 Creating all Departments' Virtual Machines

1. To create all NSA department students of both Ubuntu and Windows servers virtual machines, using the two templates: UbuS10-Temp and WinS10-Temp, can be done by the following buna cloning command:

```
root@h1:~# buna -act clone -vm UbuS10-:UbuS10-Temp WinS10-:WinS10-Temp \ 
    {NSA-{1st-vm1..10, 2nd-vm1..6}
```

Executing the above command lists the name of the new virtual machines with the path of their image file storage. It waits for users Yes/No answer to continue:

```
The generated virtual machine names from: local(qemu:///system)
The following 32 VMs file will be stored in /var/lib/libvirt/images/
UbuS10-1st-vm1 UbuS10-1st-vm2 UbuS10-1st-vm3 UbuS10-1st-vm4 UbuS10-1st-vm5
UbuS10-1st-vm6 UbuS10-1st-vm7 UbuS10-1st-vm8 UbuS10-1st-vm9 UbuS10-1st-vm10
UbuS10-2nd-vm1 UbuS10-2nd-vm2 UbuS10-2nd-vm3 UbuS10-2nd-vm4 UbuS10-2nd-vm5
UbuS10-2nd-vm6 WinS10-1st-vm1 WinS10-1st-vm2 WinS10-1st-vm3 WinS10-1st-vm4
WinS10-1st-vm5 WinS10-1st-vm6 WinS10-1st-vm7 WinS10-1st-vm8 WinS10-1st-vm9
WinS10-2nd-vm5 WinS10-2nd-vm6
```

If you agree type y:

```
If the response, from the administrator, is "Yes" then virtual machines will be created automatically:

```
root@h1:~# buna -inf all
```
```
From: local(qemu:///system)
- UbuS10-NSA-1st-vm1 shut off
- UbuS10-NSA-1st-vm10 shut off
- UbuS10-NSA-1st-vm2 shut off
```
```
Then after if information is gathered about all VMs on H1, the result is shown as follows:
```
root@h1: ~# buna -inf all
```

7.3. BUNA TESTING

As shown above, NSA department virtual machines are created; this shows that how much **buna** is simple and powerful to create number of VMs within a simple command. If someone needs to see the simulation rather than doing the actual action, it is possible to use the `-simulate` option at the end of the command.

2. To simulate the creation of 3rd year Biology and Chemistry department VMs on H2, one can use the following command.

```bash
root@h1:~# buna -con H2 -act clone -vm Win7:-Win7-Temp {Bio Chem{-3rd-vm1..12 -simulate
```

The following table shows virtual machine names with the selected path. Here, virtual machines are distributed into two storage paths.

<table>
<thead>
<tr>
<th>Waiting the administrator response</th>
<th>from: H2(qemu+ssh://192.168.0.27/system)</th>
</tr>
</thead>
<tbody>
<tr>
<td>if you agree type y:y</td>
<td></td>
</tr>
</tbody>
</table>

If the administrator types "y", then a full `virsh-clone` command of each virtual machine, that was planned to be executed, will be displayed as follows.

```
virt-clone --connect qemu+ssh://192.168.0.27/system --original Win7-Temp \n--name Win7-Chem-3rd-vm3 --file /var/lib/libvirt/images/iscsi2/Win7-Chem-3rd-vm3.img
virt-clone --connect qemu+ssh://192.168.0.27/system --original Win7-Temp \n--name Win7-Chem-3rd-vm4 --file /var/lib/libvirt/images/iscsi2/Win7-Chem-3rd-vm4.img
virt-clone --connect qemu+ssh://192.168.0.27/system --original Win7-Temp \n--name Win7-Chem-3rd-vm5 --file /var/lib/libvirt/images/iscsi2/Win7-Chem-3rd-vm5.img
```

............. .....continue .............
7.3. BUNA TESTING

3. Similarly to create Architecture department virtual machines, using the template virtual machine called *UbuD-Temp*, can be done by using the following command:

```
root@h1:~# buna -con H3 -action clone -vm UbuD-Arch-5th:UbuD-Temp {-vm1..8}
```

### 7.3.2.2 Basic Automaton Actions

1. To start three randomly selected virtual machines from all hosts can be done as follows:

```
root@h1: /final# perl buna –con all –act start –vm all -coun 3
```

From: H2(qemu+ssh://192.168.0.27/system)
... Win7-Bio-3rd-vm1:- Domain Win7-Bio-3rd-vm1 started
... Win7-Bio-3rd-vm10:- Domain Win7-Bio-3rd-vm10 started
... Win7-Bio-3rd-vm11:- Domain Win7-Bio-3rd-vm11 started

From: local(qemu:///system)
... UbuS10-1st-vm1:- Domain UbuS10-1st-vm1 started
... UbuS10-1st-vm10:- Domain UbuS10-1st-vm10 started
... UbuS10-1st-vm2:- Domain UbuS10-1st-vm2 started

From: H3(qemu+ssh://192.168.0.47/system)
... UbuD-Arch-5th-vm1:- Domain UbuD-Arch-5th-vm1 started
... UbuD-Arch-5th-vm2:- Domain UbuD-Arch-5th-vm2 started
... UbuD-Arch-5th-vm3:- Domain UbuD-Arch-5th-vm3 started
```

From the above experiments, it is seen that *buna* is powerful, flexible and simple to handle thousands of virtual machines.

2. Actions like suspend, resume, reboot, shutdown, stop and undefine can be done similarly to the previous start command. The --count option has good impact to select random number of virtual machines for any action. For example, migrating any two running virtual machines from H3 to H1 can be done using the following command:

```
root@h1: # buna -con H3 -act migrate -dest H1 -vm active -count 2 -simulate
```

```
virsh migrate --suspend UbuD-Arch-5th-vm1 qemu:///system qemu+ssh://192.168.0.47/system
virsh migrate --suspend UbuD-Arch-5th-vm2 qemu:///system qemu+ssh://192.168.0.47/system
```

3. To start all Ubuntu virtual machines of first year NSA department students:

```
root@h1: # buna -act start –vm UbuS10*1st*
```

From: local(qemu:///system)
... UbuS10-1st-vm1:- error: Domain is already active
... UbuS10-1st-vm10:- error: Domain is already active
... UbuS10-1st-vm2:- error: Domain is already active
7.3. BUNA TESTING

4. To start all inactive virtual machines of H2:

root@h1: # buna -conn H2 -act start -vm inactive

From: H2(qemu+ssh://192.168.0.27/system)
... Win7-Bio-3rd-vm3:- Domain Win7-Bio-3rd-vm3 started
... Win7-Bio-3rd-vm4:- Domain Win7-Bio-3rd-vm4 started
... Win7-Bio-3rd-vm5:- Domain Win7-Bio-3rd-vm5 started
... Win7-Chem-3rd-vm7:- Domain Win7-Chem-3rd-vm7 started
... Win7-Chem-3rd-vm8:- Domain Win7-Chem-3rd-vm8 started
... Win7-Chem-3rd-vm9:- Domain Win7-Chem-3rd-vm9 started

5. To save the CPU and Memory snapshots of 2 active virtual machines:

root@h1: # buna -con H2 -act save -vm active -count 2

From: H2(qemu+ssh://192.168.0.27/system)
Domain Win7-Bio-3rd-vm1 saved to /var/lib/libvirt/store/Win7-Bio-3rd-vm1
Domain Win7-Bio-3rd-vm10 saved to /var/lib/libvirt/store/Win7-Bio-3rd-vm10

6. To suspend the first 5 virtual machines of Chemistry department students:

root@h1: # buna -con H2 -act suspend -vm *Chem*vm{1..5

From: H2(qemu+ssh://192.168.0.27/system)
... Win7-Chem-3rd-vm1:- Domain Win7-Chem-3rd-vm1 suspended
... Win7-Chem-3rd-vm2:- Domain Win7-Chem-3rd-vm2 suspended
... Win7-Chem-3rd-vm3:- Domain Win7-Chem-3rd-vm3 suspended
... Win7-Chem-3rd-vm4:- Domain Win7-Chem-3rd-vm4 suspended
... Win7-Chem-3rd-vm5:- Domain Win7-Chem-3rd-vm5 suspended

All the above test shows that buna is capable to give the basic services for managing virtual machines.

7.3.2.3 "Update" Automation Action

To change the number of assigned CPU of 2nd year NSA department students Windows virtual machines:

root@h1: # buna -con H1 -act setvcpus -vm *Win* -count 2 -simulate

From: H1(qemu:///system)
virsh --connect qemu:///system setvcpus WinS10-1st-vm1 2
virsh --connect qemu:///system setvcpus WinS10-1st-vm10 2
7.3. BUNA TESTING

virsh --connect qemu:///system setvcpus WinS10-1st-vm2 2
virsh --connect qemu:///system setvcpus WinS10-1st-vm3 2
virsh --connect qemu:///system setvcpus WinS10-1st-vm4 2
virsh --connect qemu:///system setvcpus WinS10-1st-vm5 2
virsh --connect qemu:///system setvcpus WinS10-1st-vm6 2
virsh --connect qemu:///system setvcpus WinS10-1st-vm7 2
........continue.......

'setmem' and 'setmaxmem' can also be done as the above command by only giving the value of --count in KB.

7.3.2.4 Renaming Virtual Machines

Shifting virtual machines of all 3rd year biology students into 4th year can be done by replacing substring '3rd' with '4th' as follows:

The below result is a simulated type.

| root@h1: # buna -conn H2 -act rename --vm "Bio-3rd" -change 3rd 4th -simulate |
| From: H2(qemu+ssh://192.168.0.27/system) |
| renaming Win7-Bio-3rd-vm1 to Win7-Bio-4th-vm1 ... |
| XML change on:  <name>Win7-Bio-4th-vm1</name> |
| XML change on:  <source file="/var/lib/libvirt/images/Win7-Bio-4th-vm1.img"/> |
| ssh root@192.168.0.27 mv /var/lib/libvirt/images/Win7-Bio-3rd-vm1.img /var/lib/libvirt/images/Win7-Bio-4th-vm1.img |
| scp tempmodefiedvm.xml root@192.168.0.27:/etc/libvirt/qemu/Win7-Bio-4th-vm1.xml |
| ssh root@192.168.0.27 rm /etc/libvirt/qemu/Win7-Bio-3rd-vm1.xml |
| renaming Win7-Bio-3rd-vm10 to Win7-Bio-4th-vm10 ... |
| XML change on:  <name>Win7-Bio-4th-vm10</name> |
| XML change on:  <source file="/var/lib/libvirt/images/Win7-Bio-4th-vm10.img"/> |
| ssh root@192.168.0.27 mv /var/lib/libvirt/images/Win7-Bio-3rd-vm10.img /var/lib/libvirt/images/Win7-Bio-4th-vm10.img |
| scp tempmodefiedvm.xml root@192.168.0.27:/etc/libvirt/qemu/Win7-Bio-4th-vm10.xml |
| ssh root@192.168.0.27 rm /etc/libvirt/qemu/Win7-Bio-3rd-vm10.xml |
| renaming Win7-Bio-3rd-vm11 to Win7-Bio-4th-vm11 ... |
| XML change on:  <name>Win7-Bio-4th-vm11</name> |
| XML change on:  <source file="/var/lib/libvirt/images/Win7-Bio-4th-vm11.img"/> |
| ssh root@192.168.0.27 mv /var/lib/libvirt/images/Win7-Bio-3rd-vm11.img /var/lib/libvirt/images/Win7-Bio-4th-vm11.img |
| scp tempmodefiedvm.xml root@192.168.0.27:/etc/libvirt/qemu/Win7-Bio-4th-vm11.xml |
| ssh root@192.168.0.27 rm /etc/libvirt/qemu/Win7-Bio-3rd-vm11.xml |
| ........continue........ |

The above result shows the actions that are planned to be done for each VM name change. If --simulate is removed, then the virtual machine name will be changed as follows:
7.3. BUNA TESTING

Using the above simple way, it is possible to regroup existing virtual machines as they are needed.

7.3.3 Information Test

1. To list all running virtual machines on all hosts:

   ```
   root@h1: # buna -con all -inf running
   From: H2(qemu+ssh://192.168.0.27/system)
   - Win7-Bio-4th-vm1  shut off
   - Win7-Bio-4th-vm10 shut off
   - Win7-Bio-4th-vm11 shut off
   - Win7-Bio-4th-vm12 shut off
   - Win7-Bio-4th-vm2  shut off
   - Win7-Bio-4th-vm3  shut off
   - Win7-Bio-4th-vm4  shut off
   - Win7-Bio-4th-vm5  shut off
   - Win7-Bio-4th-vm6  shut off
   - Win7-Bio-4th-vm7  shut off
   - Win7-Bio-4th-vm8  shut off
   - Win7-Bio-4th-vm9  shut off
   - Win7-Chem-3rd-vm1 shut off
   - Win7-Chem-3rd-vm10 shut off
   - Win7-Chem-3rd-vm11 shut off
   - Win7-Chem-3rd-vm9  shut off
   - Win7-Temp        shut off
   From: local(qemu:///system)
   49 UbuS10-1st-vm1 running
   50 UbuS10-1st-vm10 running
   51 UbuS10-1st-vm2 running
   .......continue......
   From: H3(qemu+ssh://192.168.0.47/system)
   15 UbuD-Arch-5th-vm1 running
   16 UbuD-Arch-5th-vm2 running
   17 UbuD-Arch-5th-vm3 running
   18 UbuD-Arch-5th-vm4 running
   19 UbuD-Arch-5th-vm5 running
   .......continue......
   ```

2. To list paused virtual machines from Chemistry department:

   ```
   root@h1: # buna -con H2 -inf paused --vm "Chem"
   From: H2(qemu+ssh://192.168.0.27/system)
   13 Win7-Chem-3rd-vm1 paused
   17 Win7-Chem-3rd-vm2 paused
   18 Win7-Chem-3rd-vm3 paused
   19 Win7-Chem-3rd-vm4 paused
   20 Win7-Chem-3rd-vm5 paused
   ```
7.3. BUNA TESTING

7.3.4 Plugin Test

7.3.4.1 Writing Rules

Consider the university administrators plan to manage resource of these three hosts based on the following rules:

1. **Rule-1**: When the maximum number of running virtual machines on H1 is greater than 10 and in H2 is greater than 15, administrators about to be notified that "H1 and H2 are busy!''.

   ```
   $rule(H1.running.max > 10 and H2.running.max > 15; echo "H1 and H2 are busy!");
   ```

2. **Rule-2**: Consider students are using their machines occasionally; so administrators need to deploy four new virtual machines on H1, named 'admin-vm{1..4}'; where these are going to be used for some administration purpose. These virtual machines are planned to run always, unless the 75 percentile value of CPU usage of H1 is greater than 80%. That means, if the 75 percentile of CPU usage is less than 80%, these 'admin' VMs will keep running.

   ```
   $Percent= 75;
   $rule(H1.cpuusage.percentile < 80; buna -con H1 -action resume -vm admin*);
   $rule(H1.cpuusage.percentile >= 80; buna -con H1 -action suspend -vm admin*);
   ```

3. **Rule-3**: Administrators need to do load balancing between H2 and H3: If the difference of the average number of running virtual machines between H2 and H1 is greater than 6, then migrate any three running virtual machines from the host with higher VMs to the other one.

   ```
   $rule(H2.running.mean - H3.running.mean >= 6; echo migrating from H2 to H3, \$
   buna -con H2 -act migrate -vm active -count 3 -dest H3 -simulate);
   $rule(H3.running.mean - H2.running.mean >= 6; echo migrating from H3 to H2, \$
   bima -con H3 -act migrate -vm active -count 3 -dest H2 -simulate);
   ```

The result of each of the above rules is a simulation that shows the appropriate virsh migrate commands that would be executed. This is because virsh migrate command is not working in my testbed; but it is not on the problem of buna.

Consider all the rules are written in a file named 'student.rules'. The first two lines limit the sample-size and time-interval of data collection for one statistical analysis and rule validation. To make it visible, the last rule is written in two lines, but in rule file it must be finished in one line.
7.3. BUNA TESTING

Now let us run \textit{buna} as a plugin named ‘\textit{student}’ and the above rules file:

\begin{verbatim}
root@h1:~# buna -plugin student student.rules
\end{verbatim}

\subsection*{7.3.4.2 Plugin Test Results}

The results of the plugin, which is displayed based on some instances, are listed below:

1. When all virtual machines on all hosts are running, the result looks:

\begin{verbatim}
H1 and H2 are busy!
From: H1(qemu:///system)
... admin-vm1-: Domain admin-vm1 resumed
... admin-vm2-: Domain admin-vm2 resumed
... admin-vm3-: Domain admin-vm3 resumed
... admin-vm4-: Domain admin-vm4 resumed
migrating from H2 to H3
virsh migrate --suspend Win7-Bio-3rd-vm1 qemu+ssh://192.168.0.47/system
gemu+ssh://192.168.0.27/system
virsh migrate --suspend Win7-Bio-3rd-vm10 qemu+ssh://192.168.0.47/system
gemu+ssh://192.168.0.27/system
virsh migrate --suspend Win7-Bio-3rd-vm11 qemu+ssh://192.168.0.47/system
gemu+ssh://192.168.0.27/system
\end{verbatim}

The message displayed at the first line is right, because the number of running VMs on H1 is 20 and on H2 is 24. Lines between 3 and 7 show that ‘\textit{admin}’ virtual machines are resumed; this is because the CPU usage of H1 is less than 80%. Lines below 8, display three virsh-commands that are used to migrate three running virtual machines from H2 to H3.

2. When the CPU usage in H1 is higher than 80%, the plugin takes a trigger action to suspend the ‘\textit{admin}’ virtual machines. Here below is the result:
7.3. BUNA TESTING

When H1 is busy

From: H1(qemu:///system)
... admin-vm1:- Domain admin-vm1 suspended
... admin-vm2:- Domain admin-vm2 suspended
... admin-vm3:- Domain admin-vm3 suspended
... admin-vm4:- Domain admin-vm4 suspended
.....

3. Now let us see what it will be when 22 running virtual machines are stop form H2 using the following command:

stop 22 virtual machines from running

root@h1:~# buna --con H2 -act destroy -vm active -count 22

After some time, the demon start replying as follows:

No action taken by Rule-1

root@h1:~# From: H1(qemu:///system)
... admin-vm1:- Domain admin-vm1 resumed
... admin-vm2:- Domain admin-vm2 resumed
... admin-vm3:- Domain admin-vm3 resumed
... admin-vm4:- Domain admin-vm4 resumed
migrating from H3 to H2
virsh migrate --suspend UbuD-Arch-5th-vm1 qemu+ssh://192.168.0.27/system
... qemu+ssh://192.168.0.47/system
virsh migrate --suspend UbuD-Arch-5th-vm2 qemu+ssh://192.168.0.27/system
... qemu+ssh://192.168.0.47/system
virsh migrate --suspend UbuD-Arch-5th-vm3 qemu+ssh://192.168.0.27/system
... qemu+ssh://192.168.0.47/system

Rule-1, the busy message, is not replying because the number of running virtual machines on H2 is less than 15. Rule-3 condition is now triggered to migrate VMs from H3 to H2, this is because H2 has now only two running virtual machines.

4. Now by using the following command, increase the number of running virtual machines on H2 to 7; see what the demon will reply.

Running 5 virtual machines from H2

root@h1:~# buna -con H2 -act start -vm inactive -count 5

From: H2(qemu+ssh://192.168.0.27/system)
... Win7-Bio-3rd-vm1:- Domain Win7-Bio-3rd-vm1 started
... Win7-Bio-3rd-vm10:- Domain Win7-Bio-3rd-vm10 started
... Win7-Bio-3rd-vm11:- Domain Win7-Bio-3rd-vm11 started
... Win7-Bio-3rd-vm12:- Domain Win7-Bio-3rd-vm12 started
... Win7-Bio-3rd-vm2:- Domain Win7-Bio-3rd-vm2 started

The daemon start replying as follows:
7.4. SUMMERY OF BUNA TESTING

The above result shows that the action of Rule-3 is not executed, this is because the difference of the number of running virtual machines between H2 and H3 is less than 6. Rule-1 is not executed because the H1 and H2 are not busy as pre the given condition. The result shows only Rule-2 is executed, the 'admin' VMs on H1 are resumed, this is happened because the CPU usage on H1 is less than 80%.

7.4 Summery of buna Testing

The above 'Plugin part' tests has shown that by writing meaningful and evenly arranged rules it is possible to manage resources between hosts and control statuses of virtual machines in a simple and flexible manner. One can also construct rules in a way that enables the network resources run in power-saving and load-balancing modes. The test of 'Automation part' has also shown the functionalities of *buna* and its usage simplicity, flexibility and powerfulness.
Chapter 8

Discussion

System administration mainly focuses on installing, securing and maintaining software/servers and giving related supports for customers. Most of the time, such types of tasks create routine jobs for system administrators. In addition, humans are often prone to making mistakes in doing repetitive tasks. But developing programs/scripts can solve such types of tasks. From this point of view, buna is a Perl program that is developed for automating the management of virtual machines; so that to ease the challenging and repetitive tasks of system administrators.

8.1 Alternative Approaches

While developing buna, the approach was automating the functionalities of libvirt virtual machine managers called virsh, virt-manager, virt-clone, and virt-top. This is because libvirt creates a common application interface (API) to access most hypervisors and also have the plan to support more hypervisors.

But there are some already developed automated virtual machine tools. Even though they support limited hypervisors, some of them (e.g. MLN) have created an environment that is used to add new plugins to support new hypervisors. Developing plugins for MLN to support unsupported hypervisors is a feasible alternative approach.

The reason why this thesis work is based on libvirt environment than the alternative approach is to address the particular needs of some organizations like universities and training centers. Such organizations manage continually changing and large number of virtual machines, to complicate the management process. The complexity of the existing automation tools increases when they are applied in such organizations.
8.2 buna Accomplishment

*buna* has achieved its goals and in the near future the first version will be announced as an open and free tool. The approach is not difficult to implement if someone has good understanding in software design, Perl programming languages and basic CPAN modules related to system administration. Because of the limited time given for this thesis some functionalities are left and suggested for further work(section 10). Next versions of *buna* will try to accomplish that.

1. *buna* is able to create thousands of virtual machines at a time and take basic actions like start, stop, delete and migrate groups of virtual machines with simple, flexible and powerful commands. *buna* is able to change the virtual machines’ assigned CPUs, and Memory size; and attach/detach devices automatically on groups of virtual machines.

2. *buna* is able to change virtual machine names easily. This helps administrators to group/regroup virtual machines to make *buna* smart. Group of virtual machines are identified with a common string from virtual machines’ names. Then actions can be applied on a group of virtual machines by combining that common string and wild-characters(*). To apply automatic actions on the right virtual machines, administrators have to take care while giving/changing virtual machine names and referring them using group names.

3. With *buna* it is possible to manage resources between hosts. The *buna*-plugin is ready to be run as a daemon and take any action based on the fulfillment of conditions given by users. Condition evaluation will be done based on CPU and Memory usage between hosts, and number of virtual machines existing in some status like running, idle, paused, and so on. The action part can be any of *buna* automation actions, Linux command, program or script.

8.3 buna Limitations

8.3.1 Code Optimization:

Some *buna* functions are written with lots of variables and repetitive loops, which has to be optimized. For example, the *Cloning-Function*, when it checks the available storage space and evaluates it with the amount of storage space needed for new virtual machines has such problem.

8.3.2 Validation Problem

To minimize the occurrence of internal errors and miss interpretations of input values, *buna* has done lots of validation; but still it needs further analysis in the
8.3. BUNA LIMITATIONS

following points:

8.3.2.1 Validation on –vm Input Option

While using –vm option with input value which looks like '1..5', this expression might be converted into '1 2 3 4 5', by the Linux-running-shell, before it reaches to buna-name-generator. This might end with the result of wrong virtual machine names. For example '–vm Ubu:vm1 {-vm{1..5}' is expected to generate five VMs’ name called 'Ubu-vm1' 'Ubu-vm2' 'Ubu-vm3' 'Ubu-vm4' 'Ubu-vm5', but buna results only one VM name 'Ubu-vm1-vm2-vm3-vm4-vm5'. For the time being this problem is solved by giving space either between the opening or closing curl bracket and the nearby number. To get the right result, the previous example must be typed as 'Ubu:vm1 {-vm{ 1..5}' (space before number 1).

8.3.2.2 More Validation on buna Rules

1. In buna, the approach of writing rules needs further optimization. Writing rules requires users to type repetitive host-names and keywords. In some cases this may make the rule too long. Further work is needed to make rule typing simpler and shorter. One approach can be predefining variables like $a=host1.running, and then use this predefined variables inside the rules like $a.max <10.

2. While rules are converted into Perl if-condition-action code, validation is done only on the existence of used host-names and the proper use of buna property and statistics keywords. Other validations which are stated below are left for Perl itself, but it is advisable if validation is done before reaching the Perl-parser.

(a) The condition part of the rule can be written using round-brackets together with and/or Perl-keywords. But buna validation is not set to check whether they are used in a proper way or not.
(b) In the condition part, comparison symbols (=, <, <=, > and >=) are allowed but validation is not set to check whether they are used in a proper way or not.
(c) Everything collected from the action part of a rule is converted to Perl action codes, with no validation. It is better if some validation and limitations are set.
Chapter 9

Conclusion

The thesis explored an untapped area of Virtual Machine Management. It basically considered solving the challenges of System Administrators in manually managing (like creating, starting, stopping, upgrading and updating) large number of virtual machines, and resources management between hosts; especially in dynamically changing environments like universities. A novel approach based on mathematical theories is suggested and as a proof of concept a tool named *buna* is developed, deployed and tested. The tool works with most hypervisors and it automates existing libvirt based virtual machine managers.

*Buna* is a very simple-to-use, flexible and powerful command line automated management tool; which is developed in Perl program. It’s tested on a lab environment and found to be successful in automatically managing many virtual machines. It also has a plugin that can be used to manage resources between hosts as per the need of system administrators.

Using *buna*, managing virtual machines and hosts is more fun than a burden. Accordingly the thesis work has achieved its goals mentioned in section 1.5.
Chapter 10

Recommendations

Basically **buna** depends on **virsh**, **virt-clone** and **virt-top** managers. Not to be limited with only their functionalities, it is better to use the libvirt CPAN module named **Sys::Virt**.

**buna** plugin part is limited to do statistics on some properties of hosts; further work is needed to consider network and I/O usages. The statistics analysis is limited to descriptive statistics and will be better if mathematics integration and probability are also used.

Future works is needed to ground level automations on virtual machines, like install/remove software and add/delete users. It is suggested to explore a **libvirt** manager called `guestfish` as it’s capable to perform such tasks.
Bibliography


[34] Dong Yan. Automated administration of virtual machines and networks on the vmware esx platform. The advisor of this paper was, Kyrre Begnum, who is the owner of the tool that this paper did research.


Appendix A

Appendix A
MLN

MLN project file Keys and values. It is copied from the MLN manual [15].

```
[global {
  project <name of the project>
}
)
(host|superclass) <name of the host or super classes> {
  uml
  [hvm]
  [memory <memory size>M]
  [lvm]
  template <the full path of template image file>
  [size <storage size>(M|G)]
  [free_space <free additional size>(M|G)]
  [service_host <(host_ip|hostname)>]
  [superclass <superclass>]
  [boot_order <num>]
  [network <interface name> {
    [address <ip>|dhcp]
    [gateway <ip>]
    [netmask <netmask>]
    [switch <switch>]
    [bridge <br_if>]
  }]
  [users {
    [username <cryptpasswd> [uid] [homedir]
  ...]]
  }
  [groups {
    [groupname {
        [user]
      ...]
    }
  }]
  ...
  .
  .
  ...
...continue listing hosts as above...
  .
  .
  .
```

139
B. BUNA NAMING EXAMPLES

B  buna Naming Examples

Here are samples how to express names while using buna --vm option

--- University scenario eg: buna naming convention with --vm option ---

proper sets are:
OS = {Win7, Win2011-server, Ubuntu-desktop, Ubuntu-Server}
department = {NSA, engineering, Biology,...}
industry{civil, architect, mechanics, ...}
state-of-students{1st-year, 2nd-year, 3rd-year, 4th-year}

# to create 21 VMs for each biology and chemistry department
# where the template VM called win7_temp is used to be cloned:

```bash
--vm win7-:win7_temp{bio-, chem-{assistant vm1..20}
```

This line will generate names like:

win7-bio-assistant, win7-bio-vm1, ..., win7-bio-vm10
win7-chem-assistant, win7-chem-vm1, ..., win7-chem-vm10

# To create 20 VMs for each engineering department of 1st, 2nd and
# 3rd year students using win7_temp and ubuntu_temp templates

```bash
--vm win7-:win7_temp, ubuntu-:ubuntu_temp{civil, architect{-1st-vm1..20 -2nd-vm1..15 -3rd-vm1..10
```

This line will generate names like:

win7-civil-1st-vm1, ..., win7-civil-1st-vm20
win7-civil-2nd-vm1, ..., win7-civil-2nd-vm15
win7-civil-3rd-vm1, ..., win7-civil-3rd-vm10
ubuntu-civil-1st-vm1, ..., ubuntu-civil-1st-vm20
ubuntu-civil-3rd-vm1, ..., ubuntu-civil-3rd-vm10

# To create both win2010 and Ubuntu server VMs for
# each NSA department 1st, 2nd, 3rd year students:

```bash
--vm Ubuntu:temp1 Win2010:temp2{-server-{1st 2nd 3rd{-vm-{1 .. 25
```

...
# buna configuration file contains five types of variables

# First Variable is --> $USER, a common user for ssh connection to all hosts. Default is 'root'.
# Uncomment and change it if root is not used as a user.
# $USER= User Name;

# Second Variable is --> $HOST (always name, uri);
# list all servers name with their uri one by one using the above format
# the name will be used to do any task on the specific hypervisor
# default is (local, qemu:///system);
# if you need to change the default host use the name 'local' for others
$HOST = (local, qemu:///system);

# Third Variable is --> $FILE_DIR = full file path, priority;
# list available storage medias that will be used to store image files of VMs
# When one storage becomes full the system automatically change the storage path to others
# if there are more VMs going to be created.
# default storage is /var/lib/libvirt/images/
# The smaller priority number will be selected first.
$FILE_DIR = /var/lib/libvirt/images/iscsi2,1;
$FILE_DIR = /var/lib/libvirt/images/iscsi3,2;

# Fourth Variable is --> $SAVE_DIR = full directory_path
# it is a single directory or storage path used for storing snapshots (CPU and Memory)
# running VMs while save and restore actions are takes place here.
# default is /var/lib/libvirt/store/
$SAVE_DIR = /var/lib/libvirt/store/

# $DIRECTORY = /your/full/storage/directory/path;

# Fifth Variable is --> $XML = Directory_of_VMs xml file is stored
# this directory path will be refereed to handle the xml file of VMs while name change is applied default is shown below; if it's different uncomment and change the path
$XML = '/etc/libvirt/qemu/';
Listing A.2: Sample rules configuration file

```plaintext
# Sample rules file that is used during buna plugin

--- Keywords Information (not editable) ---
#Keywords going to be used together with the aliases of each URI

#RESOURCE KEY-WORDS with description (read virsh and virt--top manual)
count : number of existing vms from a host
active : number of working virtual VMs from a host
inactive : number of power-off VMs from a host
running : number of running vms form a host
blocked : number of blocked VMs from a host
paused : number of paused VMs from a host
shutdown : number of shutdown VMs from a host
shutoff : number of shutoff VMs from a host
guestmemory : amount of memory in KB used by all VMs from the host
cpuusage : percentage of CPU usage by all VMs from the host

# STATISTICS KEY-WORDS that will be used for each RESOURCES
mean : the average value of the give sample
median : the median value of given sample
mode : the more repeated value of given sample
percentile : a value where the given percent from the samplesets are under it
range : the range of sampleset
min : minimum value from sampleset
max : maximum value from sampleset

--- parameters (can be changed) ---

# "samplesize" ---> is the number of samples going to be collected about each key words. Default value is 40.
# "interval" ---> is the time frame between two samples. Default is 60 seconds
# "percent" ---> statistics percentile value for the given sample. Default is 80

#example: as per the default value the plugin will collect 40 samples with in 60s interval about each RESOURCES and does statistics based on that sample. Then execute the action part if the condition part fulfill.
# If you accept the default values mentioned above comment the following three lines
$SampleSize= 10;
$Interval= 3;
$Percent= 75;

--- default rules ---
# The following are DEFAULT RULES, uncomment them if you need to do load balancing automatically
# the number shows load balance has to be done when the memory or cpu usage is more than the given number.
```
Example user defined syntax

```bash
$rule(conditons; any action[: flag]);
```

$rule(local.running.mean <= 4; buna — action start — vm vm4);

$rule(local.running.mean > 10; buna — action destroy — vm vm4);

$rule(remote.running.mean > 10; echo overload!!!!, buna — connect remote —
action destroy — vm net*; 1);

$rule(remote.running.max > 15 or local.running.max > 8; echo overload!!!

$rule(local.running.max > 5 or remote.running.max > 8; echo overload!!!!, —
— action destroy — vm net*);

$rule(localsdfsdf.running.max > 4; perl buna — action destroy — vmg net

# $rule((local.paused.max >2 and local.inactive.max == 3) or local. —
running.max < 2; perl buna — action start — vmg so2 so1, perl buna —
— action destroy — vmg net*);
sectionbuna Perl Program This is the full Perl program of *buna* for both automation and plugin part.

Listing A.3: buna.pl

```perl
#!/usr/bin/perl
### buna: Simple, Flexible and Powerful Automated Virtual Machine and Host Management Tool Under Libvirt Environment Basic functionalities are: create, start, stop VMs, change CPU and Memory attach/detach devices on VMs and manage resource usage like CPU & Memory between their hosts and keeps Virtual Machines status as needed.

# Used Packages
use Getopt::Long;
use List::MoreUtils qw{ any uniq };
use List::MoreUtils; # for internal use
use File::Remote;

# Global Variables For Commands
@EVENT_ACTION = ('clone', 'start', 'destroy', 'suspend', 'resume', 'reboot', 'shutdown', 'save', 'restore', 'migrate', 'undefine');
@UPDATE_ACTION = ('setmem', 'setmaxmem', 'setvcpus', 'rename');
@STATE = ('running', 'idle', 'paused', 'down', 'off', 'crashed', 'dying', 'all');
@INFO = ('dominfo', 'domuuid', 'domid', 'vcpupinfo', 'ttyconsole', 'vncdisplay');
@DEVICE_ACTION = ('attach-device', 'attach-disk', 'attach-interface', 'detach-device', 'detach-disk', 'detach-interface');
@ACT_INACT = ('all', 'active', 'inactive');

# Default Values
$USER = 'root'; # for ssh connection to all hosts
$HOST = ('local', 'qemu:///system'); # default host
$config_file = '/etc/buna/buna.conf';
$skeleton_program = '/etc/buna/skeleton'; # skeleton for plugin part
$file_dir_list = [('/var/lib/libvirt/images/')]; # for VMs image file
$directory = '/var/lib/libvirt/store/'; # for snapshot
$xml = '/etc/libvirt/qemu/'; # XML file storage directory for VMs

# Reading from configuration file to update default values
open(MYFILE, "$config_file") or die "Unable to open \"$config_file\": $!\n";
while($line=<MYFILE>){
    chomp $line;
    $USER =~ s/\s*/\$USER\s*\=\$USER\s*;/;
    $HOST =~ s/\s*/\$HOST\(\(\.\)\)+\(\(\.\)\)+\)/;
    $FILE_DIR_LIST =~ s/\s*/\$FILE_DIR\(\(\.\)\)+\(\(\.\)\)+\)/;
    $directory =~ s/\s*/\$directory\(\(\.\)\)+\(\(\.\)\)+\)/;
}$
```

E. BUNA.PL
```perl
38  $XML= $1 and next if ($line =~ /^\s*(XML|)\s*=\s*(.+)?\s*;/);
39
close ($MYFILE);
40  uniq @FILE_DIR_LIST;
41
42
43
44
45
46  # Handling Flags and Arguments
47  GetOptions ( "help!" => \$help,  
48      "simulate!" => \$simulate,  
49      "plugin: s{.}" => \@plugin,  
50      "vm: s{.}" => \@step,  
51      "count: i" => \$count,  
52      "information: s" => \$info,  
53      "action: s" => \$action,  
54      "live!" => \$live,  
55      "connect: s{.}" => \@connhost,  
56      "destination: s" => \$desthost,  
57      "directory" => \$DIRECTORY,  
58      "xml" => $XML,  
59      "change: s{.}" => \@name,  
60      "remove" => \$remove_vms_image,  
61      "device: s{.}" => \@device
62  ) ;
63
64  # Validation Area
65  if ( $help ) {
66    usage();
67    exit 0;
68  }
69
70  unless ( \@plugin or $info or $action or \@device ) {
71    print "--help, --plugin, --information, --action or --device value is mandatory
";
72    exit 0;
73  }
74
75  if ( \@connhost ) {
76    if ( $connhost[0] eq 'all' ) {
77      \@connhost = keys %HOST;
78    } else {
79      foreach $connhost ( \@connhost ) {
80        unless ( $HOST{ $connhost } ) {
81          print "no uri with aliases: $connhost\n";
82          listhosts( $HOST );
83        }
84      }
85    }
86  } else {
87    $connhost[0] = 'local';
88  }
89
90  $connhost = $connhost[0];
91
92  $CONNECT_URI = $HOST{ $connhost };
```
if ($CONNECT_URI =~ /\^.*\\.+\\/.*\\.*$/) {
    $connip = $1;
    if ($connip) {
        $full_user_name[0] = "ssh USER " . '@' . $connip;
        $full_user_name[1] = "scp USER " . '@' . $connip . ': ';
    }
    else {
        @full_user_name = "";
    }
}

$SOURCE_URI = $HOST { $connhost; }
unless ($SOURCE_URI) {
    print "no uri with aliases: $connhost\n";
    listhosts($HOST);
}

if ($desthost) {
    $DESTINATION_URI = $HOST { $desthost; }
    unless ($DESTINATION_URI) {
        print "no host with \"$desthost\" name\n";
        listhosts($HOST);
    }
    else {
        $desthost = 'local';
        $DESTINATION_URI = $HOST { $desthost; }
    }
}

$DIRECTORY = $DIRECTORY . '/ ' unless ($DIRECTORY =~ /*\$/);
$XML = $XML . '/ ' unless ($XML =~ /*\$/);
$step = '@step' if (@step);
chomp $step;
chomp $SOURCE_URI;
chomp $DESTINATION_URI;

# Type-Checker
if ($info) {
    Sub_Information();
} elsif ($action eq 'rename') {
    Sub_Rename();
} elsif ($action eq 'clone') {
    Sub_Clone();
} elsif ($action eq 'migrate') {
    Sub_Migrate();
} elsif ($action eq 'save') {
    Sub_Save();
} elsif ($action eq 'restore') {
    Sub_Restore();
} elsif ($action eq 'undefine') {
    Sub_Undefine();
} elsif (any {$_ eq $action} @UPDATE_ACTION) {
    Sub_Update_Action();
} elsif (any {$_ eq $action} @EVENT_ACTION) {
    Sub_Event_Action();
}
```perl
149 } elsif ($action) {
150     # print usage
151     print "---action values are : -n";
152     print "@EVENT_ACTION\n";
153     print "@UPDATE_ACTION\n";
154     exit 0;
155 } elsif (@device) {
156     Sub_Device_Action();
157 } elsif (@plugin) {
158     Sub_Plugin();
159 } else {
160     print "unknown error\n";
161     usage;
162     exit 0;
163 }
164
165 # Information Collector
166 sub Sub_Information{
167     my @lines;
168     my $command;
169     my @templines;
170     # Gathering the status of VMs like running, idle, crashed, ...., and all
171     if (any { $_ eq $info } @STATE) {
172         foreach $connhost (@connhost) {
173             @full_user_name = Sub_ssh_user($connhost); # adjust for ssh ->
174             # gathering information with out VMs
175             if ($info eq 'all') {
176                 $command = 'virsh --connect ' . "'CONNECT_URI list --all";
177                 open(MYFILE, "$command |") or die "$command, $!";
178                 @lines = <MYFILE>;
179                 splice @lines, 0, 2;
180             } else {
181                 $command = 'virsh --connect ' . "'CONNECT_URI list --all";
182                 open(MYFILE, "$command |") or die "$command, $!";
183                 @lines = <MYFILE>;
184             }
185         }
186         # Gathering information for the given VMs
187         if ($step) {
188             my @templines;
189             @ms-vm_syntax_filter($step); # call VMs name generator
190             foreach $line (@lines) {
191                 foreach $vm (@ms) {
192                     if ( $line =~ /$vm/) {
193                         push @templines, $line;
194                     }
195                 }
196             }
197         }
198     } else {
199         @lines = uniq @lines;
200     }
201     @lines = uniq @lines;
```
```perl
204  @lines = @lines[0 .. $count - 1] if ($count > 0 and $count < @lines);
205  print " @lines\n";
206  }
207  elsif (any { $_.eq $info } @INFO) {
208     # Gathering detail information about VMs
209     print ""--vm is mandatory\n" and exit 0 unless (@step);
210     # adjusting ssh connection for hosts
211     foreach $connhost (@connhost) {
212         @full_user_name = Sub_ssh_user($connhost); # adjust for ssh connection
213         @vm_syntax_filter($step); # call VMs name generator
214         @vm = @vm[0 .. $count - 1] if ($count > 0 and $count < @vm);
215         chomp($action);
216         foreach my $vm (@vm) {
217             print "$vm ...\n";
218             $command = 'virsh --connect '."$CONNECT_URI $INFO $vm"
219             ;
220             if ($simulate) {
221                 print "$command\n";
222             } else {
223                 system("$command");
224             }
225         }
226     }
227     # printing available information values
228     print "option --state values are:\n";
229     print "$STATE\n";
230     print "$INFO\n";
231     exit 0;
232 }
233 }
234 # Changing Virtual Machines' name
235 sub Sub_Rename{
236     print "--vm is mandatory\n" and exit 0 unless (@step);
237     if ($name[0] !=) {
238         print "replacing \'$name[0]\' with \'$name[1]\' on virtual machine names\n";
239         } else {
240             print "--change <old-name-part> <new-name-part> is not given \n"
241             ;
242             exit 0;
243         }
244     }
245     foreach $connhost (@connhost) {
246         @full_user_name = Sub_ssh_user($connhost); # adjust for ssh connection
247         @vm_syntax_filter($step); # call VMs name generator
248         @vm = @vm[0 .. $count - 1] if ($count > 0 and $count < @vm);
249         my $temp_modified_vm_xml = "$tempmodifiedvm.xml";
250         foreach my $vm (@vm) {
251             $new_vm_name = $vm;
```
new_vm_name =~ s /$name[0]/$name[1]/g;

next if ($vm eq $new_vm_name);
p=print "renaming $vm to $new_vm_name ...
";
$command = "virsh --connect $CONNECT_URI dumpxml $vm";
my @old_xml_file = $command;
exit 0 unless (@old_xml_file);

my @old_file_path = grep (/source file=/, @old_xml_file);
if ($old_file_path[0] =~ / (.*) /) {
    @old_file_path[0] = $1;
} else {
    print "$old_file_path\\n";
    print "$libvirt_virtion problem\\n";
    exit 0;
}

if ( $old_file_path =~ / (.*) (\..*) /g) {
    $new_file_path = $1.$new_vm_name.$3;
    $old_xml_path = $XML.$vm.'xml';
    $new_xml_path = $XML.$new_vm_name.'xml';

open (NEWXML, '<>$temp_modified_vm_xml') or die "Unable to open temporary file, may be privilege problem": $!

foreach my $var (@old_xml_file) {
    if ($var =~ /<name>$vm</name>/) {
        $var = "<name>$new_vm_name</name>\\n";
    } elsif ($var =~ /<source file=.*>/) {
        $var = "<source file="$new_file_path' />\\n";
    } print "XML change on: $var" if ($simulate);

} unless ($simulate) {
    print NEWXML "$var";
}

close(NEWXML);

$command= "$full_user_name[0] mv $old_file_path $new_file_path";
$command1= "$scp $temp_modified_vm_xml $full_user_name[1] $new_xml_path";
$command2= "$full_user_name[0] rm $old_xml_path";

if ($simulate) {
    print "$command\\n";
    print "$command1\\n";
} else {
    system("$command");
    system("$command1");
    system("$command2");
}  
}  
} 

if ($simulate) {
    print "$command3\n"
    print "$command4\n"
} else {
    system("$command3");
    system("$command4");
}

# Cloning Automation
sub Sub_Clone {
    print "--- vm is mandatory\n" and exit 0 unless (@step);

    templateVM:
    $templateString=$step;
    print " from: $connhost($CONNECT_URI)\n";

    # Identifying the given template VMs from ---vm value
    if ($step =~ /:/) {
        my $template;
        my $group;
        while (1) {
            if ($templateString =~ /([^a-zA-Z0-9]+)\s*:\s*([^a-zA-Z0-9<]+)/) {
                $group=$1;
                $template=$2;
                chomp $group;
                chomp $template;
                $templateVM{$group}[0]=$template;
                chomp $2:
                $command = "virsh --connect $CONNECT_URI dumpxml $templateVM[$group][0]";
                $file_path=$command ;
                chomp $file_path ;
                unless ($file_path){
                    print "failed to get template \"$template\".\n";
                    exit 0;
                }
            }
        }
    }

    $command="$full_user_name[0] du $file_path | awk \"{print-\"$1\"}\"";
    my $temp_size="$command";
    chomp $temp_size;

    unless ($temp_size =~ /d+/){
        print "no image file for \"$template\" template.\n";
        exit 0;
    }
}
$templateVM[$group][1] = $temp_size;
$templateString =~ s/$group:$template/ / ;
} else {
  last;
}

$step =~ s/\s*:\s*\[a-zA-Z_0-9-]+//g;
if ($step =~ /:/) {
  print "Wrong template VM naming or improper use of ':', see ←−vm value

  exit 0;
} else {
  print "No template VM is given

  exit 0;
}

# Checking the available storage medias from the given host
$command = "$full_user_name[0] df / awk '{print $6 \"," "$4}\'}" ;
@free_space = `$command` ;
foreach $line (@free_space) {
  ($path, $freeKB) = split (\"\\", $line);
  chomp $path;
  chomp $freeKB;
  $df_result{$path} = $freeKB;
}

# Checking where this file systems storage media is
$temp_arr = ();
$i = 0;
foreach $file_dir (@FILE_DIR_LIST) {
  if ($file_dir) {
    $file_dir = $file_dir . "/" unless ($file_dir =~ /\//);
    $temp_arr[$i] = $file_dir;
    $i++;
  }
}
@FILE_DIR_LIST = @temp_arr;
foreach $file_dir (@FILE_DIR_LIST) {
  $file_dir_is{$file_dir} = "/" ;
  foreach $path (keys %df_result) {
    my $x = $path;
    $x = s/\///\///g;
    if ( $file_dir =~ /$x\//) {
      if ( length($path) > length( $file_dir_is{$file_dir} )){

        $file_dir_is{$file_dir} = $path;
        
      }
    }
  }
}
%filter_dir=();

foreach $file_dir ( keys %file_dir_is) {
    @{$filter_dir{$file_dir_is}{$file_dir}}= ($file_dir, $df_result{←$file_dir_is}{$file_dir});
}

# Informing the user that where VMs are going to be stored from the given storage files and which VMs will not be created because of no free space
@vm-vm_syntax_filter($step); # call VMs name generator
my $size=0; # increment of storage size of each VM
my $count=0; # counting VMs
my @cross=(keys %templateVM);
my $group=join(' | ', @cross);
my @groupedVms = grep /($group)/, @vms;
@vms= @groupedVms;

foreach $path ( keys %filter_dir) {
    $size=0;
    $full=0;
    my @thispathvms;
    while ($count <= $#vms) {
        foreach $group (keys %templateVM) {
            if ($vms[$count] =~ /.*$group.*/) and ($group){
                if ($filter_dir{$path}[1] > ($size + $templateVM{←$group}[1])){
                    push @thispathvms, $vms[$count];
                    $size= $size + $templateVM{$group}[1];
                    last;
                } else {
                    $full=1;
                    last;
                }
            } else {
            }
        }
    }
    last if ($full);
    $count++;
}

my $i=@thispathvms;
print "$The following $i VMs file will be stored in $filter_dir{←$path}[0]:\n"
@$stack{ $filter_dir{←$path}[0]}= @thispathvms;
print " @thispathvms\n";
}

if ($count <= $#vms) {
    my $i= @vms - $count;
    print "\nWARNING no free space for the following $i VMs!!\n"
    print "@vms[$count .. $#vms]\n"
}

# Creating VMs
print " if you agree type y:";
my $check=<>;
chomp $check;
if($check eq 'y') {
    foreach $path (keys %stack) {
        system("$full_user_name[0] test -d $path");
        my $rc = $? >> 8;
        if ($rc) {
            print "creating new directory: $path\n";
            system("$full_user_name[0] mkdir -p $path");
            system("$full_user_name[0] chmod 777 $path");
        }
    }
    foreach my $vm (@stack($path)) {
        $file_path = "$path" . $vm . ".img";
        my $original;
        foreach $group (keys %templateVM) {
            if ($vm =~ /.*$group.*$/) and $group) {
                $original = $templateVM{$group}[0] ;
                last;
            }
        }
        my $command = 'virt-clone --connect ' . "$CONNECT_URI ← original $original --name $vm --file $file_path" ;
        if ($simulate) {
            print "$command\n";
        } else {
            system("$command");
        }
    }
}

sub Sub_Migrate{
    print "−−vm is mandatory\n" and exit 0 unless (@step);
    if($SOURCE_URI eq $DESTINATION_URI) {
        print "migration: with the same source−−connect and −−←
        destination not possible\n";
        exit 0;
    }
    @vm = @vm_syntax_filter($step) ; # call VMs name generator
    foreach my $vm (@vm) {
        if ($live) {
            $command = "virsh migrate --live " . "$vm $DESTINATION_URI ← $SOURCE_URI" ;
        } else {
            $command = "virsh migrate --suspend " . "$vm $DESTINATION_URI ← $SOURCE_URI" ;
        }
        if ($simulate) {
            print "$command\n";
        } else {
            system("$command");
        }
    }
# Saving Action (takes snapshot of running VMs)
sub Sub_Save{
  print "−−vm is mandatory\n" and exit 0 unless (@step);
  foreach $connhost (@connhost) {
    @full_user_name = Sub_ssh_user($connhost); # adjust for ssh connection
    system("$full_user_name test -d DIRECTORY");
    my $src = $? >> 8;
    if ($src) {
      print " creating new directory: DIRECTORY\n";
      system("$full_user_name mkdir -p DIRECTORY");
    }
  }
  @vms = vm_syntax_filter($step); # call VMs name generator
  @vms = @vms[0..$count - 1] if ($count > 0 and $count < @vms);
  foreach my $vm (@vms) {
    $s = "DIRECTORY" . "$vm";
    my $command = 'virsh --connect '. "$CONNECT_URI save $vm $s";
    if ($simulate) {
      print "$command\n";
    } else {
      system ($command);
    }
  }
  exit 0;
}

# Restoring Action (run VMs from their previous snapshot)
sub Sub_Restore{
  print "−−vm is mandatory\n" and exit 0 unless (@step);
  # adjusting ssh connection to hosts
  foreach $connhost (@connhost) {
    @full_user_name = Sub_ssh_user($connhost); # adjust for ssh connection
    system("$full_user_name test -d DIRECTORY");
    my $src = $? >> 8;
    if ($src) {
      print "incorrect --stordir path : DIRECTORY\n";
      exit 0;
    }
  }
  @vms = vm_syntax_filter($step); # call VMs name generator
  @vms = @vms[0..$count - 1] if ($count > 0 and $count < @vms);
  foreach my $vm (@vms) {
    $s = "DIRECTORY" . "$vm";
    system("$full_user_name test -d $vm");
    my $rc = $? >> 8;
    if ($rc) {
      print "$vm not already saved\n";
    } next;
  }
  exit 0;
}
my $command = 'virsh --connect ' . "$CONNECT_URI restore $s";

if ($simulate) {
    print "$command\n";
} else {
    system("$command");
}
exit 0;

# undefining VMs and removing their image file
sub Sub.Undefines {
    print "--vm is mandatory\n" and exit 0 unless (@step);
    foreach $connhost (@connhost) {
        @full_user_name = Sub_ssh_user($connhost); # adjust for ssh connection
        @vm_names = vm_syntax_filter(@step); # call VMs name generator
        @vm_names = @vm_names[0 .. $count - 1] if ($count > 0 and $count < @vm_names);
        chomp($action);
        # undefine VMs
        foreach my $vm (@vm_names) {
            chomp($vm);
            if ($vm) {
                $command = 'virsh --connect ' . "$CONNECT_URI undefine $vm";
                if ($simulate) {
                    print "$command\n";
                } else {
                    if ($remove_vms_image) {
                        print "removing image file ....\n";
                        $command1 = "virsh --connect $CONNECT_URI --dumpxml $vm | grep '<source file=' | awk -F" " '{ print \$2 }";
                        $file_path = $command1;
                        chomp $file_path;
                        print ".......... $file_path \n";
                        system("$full_user_name[0] rm -r $file_path");
                    }
                    system("$command");
                }
            }
            exit 0;
        }
    }
    exit 0;
}

# other common actions that run with the same command like start, destroy
sub Sub.Event_Action {
    # same command type actions
# if (any { $_ eq $action } @EVENT_ACTION) {
print "−−vm is mandatory\n" and exit 0 unless (@step);
# adjusting ssh connection for hosts
foreach $connhost (@connhost) {
# adjusting ssh connection for hosts
    @full_user_name= Sub_ssh_user($connhost); # adjust for ssh ←
    @ms= vm_syntax_filter($step); # call VMs name generator
    @ms = @ms[0 .. $count - 1] if ($count > 0 and $count < @ms);
    chomp ($action);
    # generate command and take action
    foreach my $vm (@ms) {
        chomp ($vm);
        if ($vm) {
            my $command = 'virsh --connect ' . "$CONNECT_URI $action $vm←$count";
            if ($simulate) {
                print "$command\n";
            } else {
                system("$command");
            }
        }
        $vm= ' ';
    }
}
# Update CPU and memory on the fly
sub Sub_Updat_Action{
print "−−vm is mandatory\n" and exit 0 unless (@step);
print "−−count : number of cpu is needed\n" and exit 0 unless ($count)←
; # adjusting ssh connection to hosts
foreach $connhost (@connhost) {
    @full_user_name= Sub_ssh_user($connhost); # adjust for ssh ←
    @ms= vm_syntax_filter($step); # call VMs name generator
    # updating action
    foreach my $vm (@ms) {
        my $command = 'virsh --connect ' . "$CONNECT_URI $action $vm←$count";
        if ($simulate) {
            print "$command\n";
        } else {
            system("$command");
        }
    }
    exit 0;
}
# Attach and Detaching devices for VMs
sub Sub_Device_Action{
print "−−vm is mandatory\n" and exit 0 unless (@step);
# check device action type
if (any {$_ eq "$device[0]"} @DEVICE_ACTION) {
  # adjust ssh connection for host
  foreach $connhost (@connhost) {
    @full_user_name=Sub_ssh_user($connhost); # adjust ssh connection
    @vms=vm_syntax_filter($step); # call VMs name generator
    splice @device, 1, 0, $vms;
    # adjust ssh connection for host
    foreach my $vm(@vms) {
      $vm=splice @device, 1, 1, $vm;
      # splice @device, 0, $vm; $device="@device";
      $command='virsh --connect ' . "$CONNECT_URI $device" ;
      if ($simulate) {
        print "$command \n" ;
      } else {
        system("$command");
      }
    }
  }
} else {
  # print device usage for user
  print "−−device needs all mandatory values except VM name as follows:\n" ;
  print " attach−device [XML−of−the−device]\n" ;
  print " detach−device [FILE]\n" ;
  print " attach−disk [SOURCE−file−access−path] [TARGET−device−access−path]\n" ;
  print " attach−disk [TARGET−of−the−device]\n" ;
  print " attach−interface [TYPE−network−or−bridge] [SOURCE−device]\n" ;
  print " detach−interface [TYPE−network−or−bridge]\n" ;
  print " see virsh manual or this document for detail \n understand\n" ;
  exit 0;
}
# Plugin code generator, generates on the fly script for the plugin
sub Sub_Plugin{
  @property=( 'count', 'running', 'blocked', 'paused', 'shutdown', '← shutoff', 'crashed', 'active', 'inactive', 'noofcpu', 'cpusage←物理memory', 'guestmemory');
  @stat=(mean,median, mode, percentile, range, min, max);
  $RULE_FILE="/etc/buna/buna.rules" unless ($RULE_FILE);
  $PLUGIN_NAME="s/[^a-zA-Z_0-9-\-]//g;";
  # assigning default values
  $SampleSize= "40;";
}
$Interval = "60";
$Percent = "80";
$LineCount = 1;

# Validating rules, filtering names of used hosts and defined ←
# variables' values
my %RuleHost;
my @CancelledRules;

open(MYFILE, "$RULE_FILE") or die "Unable to open \$RULE_FILE\" : $!

while($line=<MYFILE){
    chomp $line;
    # reading variable value to update the default
    $SampleSize = "$1;" if ($line =~ /$SampleSize\s*\=\s*(\d+)\s*;\s*/);
    $Interval = "$1;" if ($line =~ /$Interval\s*\=\s*(\d+)\s*;\s*/);
    $Percent = "$1;" if ($line =~ /$Percent\s*\=\s*(\d+)\s*;\s*/);

    # check the syntax of rules(host, property and statistics ←
    # keywords)
    if ($line =~ /\s*\$rule\s*\=\s*\(\s*\$\w+\s*\)\s*/)
        my @rule = split(\', ', $1);
        while($rule[0] =~ /\w+/)
            if (any {$_ eq $2} @property and any {$_ eq $3} ←
                @stat) {
                $RuleHost{$1}=$HOST;$1:
            } else {
                print "WARNING!!! property/statistics problem → ←
                \$2, \$3\n";
                print "$line\n";
                push @CancelledRules, $line;
            } else {
                print "WARNING!!! host problem→$1, see the following←
                rule\n";
                print "$line\n";
                push @CancelledRules, $line;
            }
        }
    } else {
        $rule[0] =~ s/^\$\w+\s*\$\w+\s*\$\w+//;
    }
    $LineCount++;
}

close(MYFILE);

if (@CancelledRules){
    print "if you need to continue, type y: ";
    my $check=<>;
    chomp $check;
    if ($check ne 'y') {
        print "\n\n\nprocess aborted!!!\n\n\n";
        exit 0;
    }
}
$hostlist="";
foreach $cur ( keys %RuleHost) {
  my $key = \$\cur . \"\';
  my $value = \$\HOST\\$\cur . \"\';
  $hostlist = $hostlist . $key . $value;
}

$hostlist =~ s /,\s*/ / ;

# Taking the skeleton program to the new plugin script
open (SKELETON "$SKELETON_PROGRAM") or die "Unable to open \'$SKELETON_PROGRAM\';
SKELETON_PROGRAM;
close (SKELETON);
# Writing the new plugin script
open (PLUGIN "$PLUGIN_NAME") or die "Unable to open \$PLUGIN_NAME\';
PLUGIN_PROGRAM;
foreach $PLUGIN_PROGRAM ($PLUGIN_PROGRAM) {
  print PLUGIN "$SKELETON_PROGRAM";
  # Variable space here
  if ($SKELETON_PROGRAM =~ /\s*\# Begin_Variable\s*/) {
    print PLUGIN "\$SampleSize = $SampleSize\n";
    print PLUGIN "\$Interval = $Interval\n";
    print PLUGIN "\$Percent = $Percent\n";
    print PLUGIN "\% hostlist = ($hostlist) ;\n"
  }
  if ($SKELETON_PROGRAM =~ /\s*\# Begin_Rule\s*/) {
    $LineCount=0;
    $rulecount=0;
    # Changing each rule to a code
    open (MYFILE "$RULE_FILE") or die "Unable to open \'\$RULE_FILE\';
    while ( $line =< MYFILE ) {
      # rule space here
      chomp $line;
      if (any { $_ eq $line } @CancelledRules) {
        next;
      }
      @onetime;

      if ($line =~ /\s*\$rule\(\s*\(\s*\)$rule\(\s*\)\s*\)\s*\)/) {
        $rulecount++;
        @rule = split ("\", $1);
        $rule[0] = s/(\w+)/\$image\$\r\n\1\$string))/g;
        print PLUGIN "if ($rule[0]) \n";
        @order = split ("\", $rule[1]);
        print PLUGIN "unless ($onetime\$rulecount eq \'yes\') \n";
        if ($rule[2]) {
          foreach $order (@order) {
            $command = 'system( \' . \"$order\" )';
          }
        }
      }
    }
  }
}
838  print PLUGIN "Scommand\n";
839  }
840  print PLUGIN "\n$ sometimeSrulecount='yes ':\n" if ($rule[2]);
841  print PLUGIN "\n" if ($rule[2]);
842  }
843  } if ($rule[2]);
844  }
845  print PLUGIN "\n"
846  } if ($rule[2]);
847  }
848  print PLUGIN "\n"
849  if ($line =~ /\s*\$LoadBalance\((\s*(.+))\s*\)/) {
850  ($sh1, sh2, $balancetype, $stolerate) = split(':', $1);
851  print PLUGIN "\nLoadBalance($sh1, $sh2, $balancetype, $stolerate);"
852  }
853  $lineCount++;  
854  }
855  close (PLUGIN);
856  # Start running the new plugin
857  system("chmod 777 $PLUGIN_NAME");
858  system("perl $PLUGIN_NAME &");
859  exit 0;
860  }
861  # Cloning Action
862  
863  # VMs name generator (extract the exact and full name of VMs using
864  # --vm value)
865  sub vm_syntax_filter |  
866  # separate cross product limit
867  my @vms_all;
868  my @cross=split('}', $_[0]);
869  foreach my $cross (@cross)
870  next if ($cross eq '');
871  my @vms_cross=1;
872  # separate sets in a cross product range
873  my @sets=split('}', $cross);
874  foreach my $set (@sets)
875    chomp $set;
876  next if ($set eq '');
877  # extracting sequential numbers
878  while (1) {
879    if ($set =~ /\([\^{}:\s]+\d+\s+\]?\{\^\}\:\d+\)?\d+/)
880      {
881      my $pre = $1;
882      my $min=$2;
883      my $max=$4;
884      my $post = $5;
885      if ($min <= $max)
886        my $t=$min;
```perl
my $min=$max;
$max=$t;
    }
    my $text:
    foreach my $i($min..$max){
        $text="$text " . $pre . $i . $post;
    }
    $set =~ s/([^:\s]*\s?)(\d+)\s+(\d+)/([^:\s]*\s*)/g ↔ $text/;
}
else{
    @set_expand = split( '\s', $set);
    @temp=@vms_cross;
    @vms_cross=();
    # cross product between sets to get full name
    foreach my $Temp(@temp){
        if ($Temp){
            if ($Temp =~ /\./){
                push @vms_cross, $Temp;
            } else{
                foreach my $set_expand(@set_expand){
                    if ($set_expand){
                        my $string=$Temp . $set_expand;
                        push @vms_cross, $string;
                    }
                }
            }
        }
    }
    push @vms_all, @vms_cross;
}
my @semi_final;
    if ($action eq 'clone'){# removing all non-alphanumeric characters inside the name
        my $i = 0;
        foreach my $vm(@vms_all){
            $vm =~ s/[a-zA-Z0-9\-\s]//g;
            if ($vm){
                $semi_final[$i] = $vm;
                $i++;
            }
        }
    } else{
        # removing non-alphanumeric characters inside the name except *
        my $i = 0;
        foreach my $vm (@vms_all){
            $vm =~ s/[a-zA-Z0-9\-\s]//g;
        }
    }
```
E. BUNA.PL

```perl
	$vm =~ s/^d/d/;
	if ($vm)
		$semi_final[$i] = $vm;
	$i++;
	}
	}
	
	@semi_final = uniq @semi_final;
	
	if (@semi_final == 0)
	  # print usage to the user
	  print "\nVMs have to follow the rule: Group_Name:template←vm←name1\nvm2\nvm3\n;"
	  print "the simplest rule to create a vm called vml:template←vm←name\n;"
	  print "to create number of vms that have vm at the beginning vm←template\nadmin 1..10\n;"
	  print "more complex vms Common1:template1 ... CommonN:templateN←\n|←semi–Common1 ←semi–Common2{←admin –vm10..25}\nGroup3←\ntemplate3\njohn piter 1..10\n;"
	  print "to refer existing vms use *, ex ←vm Group* or ←vm vsm*\n;"
	  print "read the manual for detail understanding\n";
	  exit 0;
	}
	
	if ($action eq 'clone')
	  return @semi_final;
	else
	  my $actinact = '@semi_final';
	  # filter existing VMs based on active and inactive cases, and ←
extracted names
	  if (any { $_ eq $actinact } @ACT_INACT)
	    if ($semi_final[0] eq 'active')
	      $command = 'virsh --connect ' . "$CONNECT_URI " . 'list←
|
awk '{ print $2}'';
	    } elsif ($semi_final[0] eq 'inactive')
	      $command = 'virsh --connect ' . "$CONNECT_URI " . 'list←
|←inactive | awk '{ print $2}'';
	    } elsif ($semi_final[0] eq 'all')
	      $command = 'virsh --connect ' . "$CONNECT_URI " . 'list←
|←all | awk '{ print $2}'';
	  }
	  open(MYFILE, "$command 1");
	  @lines = @MYFILE;
	  close(MYFILE);
	  splice @lines, 0, 2;
	  foreach $line (@lines)
	    chomp ($line);
	    push @vms, $line if($line);
	  }
	  @vms = uniq @vms;
	  my $j = 0;
	  my @remove_empty;
	  foreach my $vm (@vms)
	    chomp $vm;
	    if ($vm)
```

162
```perl
@semi_final =~ s/\*/*\[a-zA-Z0-9_\-\]/\*/g;
foreach $line (@lines) {
    chomp ($line);
    if ($line =~ /^[^.]+$/i) {
        push @ms, $1 if ($1);
    }
}
@ms = uniq @ms;
my $j = 0;
my @remove_empity;
foreach my $vm (@ms) {
    chomp $vm;
    if ($vm) {
        $remove_empity{$j} = $vm;
        $j++;
    }
    @ms = uniq @ms;
}
print "No VM with \'$step\' name\n" and exit 0 if (@ms == 0);
return @ms;
```

```perl
sub listhosts {
    my %host = %_
    print "\nexisting hosts are:\n";
    foreach my $key (keys %host) {
        print "$key\" \"\" \"\"Host\$key\"\n";
    }
    return @ms;
}
```

print "if it is new host name add it in configuration file \n SCONFIG_FILE/\n"
exit 0;
}

# Preparing for ssh connections
sub Sub_ssh_user{
my $connhost =$_[0];
@full_user_name = 
$CONNECT_URI = HOST/$connhost;
if ($CONNECT_URI =~ /^\+/\+(\+)?\+(\+)?$/) {
  $connip =$_[1];
  if ($connip) {
    $full_user_name[0] = "ssh USER" . '@' . $connip;
    $full_user_name[1] = "USER" . '@' . $connip . ':';
  } else {
    @full_user_name = 
  }
}
print "From: $connhost($CONNECT_URI)\n"
return @full_user_name;
}

# Check and reate file or directory
sub FileDirCheck{
my $command = $_.[0];
my $type = $_.[1];
my $path = $_.[2];
system $command, "$type", $path;
my $rc = $? >> 8;
if ($rc) {
  print "file $file doesn't exist on $host\n"
} else {
  print "file $file exists on $host\n"
}
exit 0;
}

# Usage of this tool
sub usage{
print "\t buna [\t --connect <Host-list>] [\t \t \t \t \n"
print "\t [\t --<information \t action \t device \t plugin> <command>] [\t \n"
print "\t [\t --vm <VM-name-list>] [\t \t \n"
print "\t [\t --destination <dest-host>] [\t \t \n"
print "\t [\t --directory <full-path>] [\t \t \n"
print "\t [\t --count [\t <memory-size \t number-of-cpu \t number-of-VMs]\t\n"
print "\t [\t --<live \t remove \t help>]]\n"
print "\t to print this usage :tbuna --help\n"
print "\t --simulate(-s) : is used to simulate actions\n"
print "\t to run the plugin:\n"
print "\t buna --plugin <PLUGIN-NAME RULE-FILE>\n"
}
prin "to get information:\n";
print "\t\tbuna —information <INFO—VALUES>
print "\t\t—connect <LIST—OF—HOSTS>
print "\t\t—vm <VMs—name>
print "\t\t—count <number—of—VMs | memory—size—in—KB | number—of—CPUs>
print "to take action:\n";
print "\t\tbuna —action <ACTION—VALUES>
print "\t\t—connect <LIST—OF—HOSTS>
print "\t\t—vm <VMs—name>
print "\t\t—change <The—Old—String—From—VMs—Name> <A—New—String—For—VMs—Name>
print "\t\t—count <number—of—VMs | memory—size—in—KB | number—of—CPUs>
print "to attach/detach device:\n";
print "\t\tbuna —device <DEVICE—VALUES>
print "\t\t—vm <VMs—name>
print "\t\t—VMs—name>
print "VALUES:\n"
print "—plugin <PLUGIN—NAME RULE—FILE>
print "\tPLUGIN—NAME : the name of the plugin script when it runs as a daemon\n";
print "\tRULE—FILE : full path of rule file, default is '/etc/buna/buna.rules'
"
print "INFO—VALUES:\n"
print "\trunning idle paused down off crashed dying all\n";
print "\tdominfo domuuid domid vcpinfo ttyconsole vncdisplay\n"
print "ACTION—VALUES:\n"
print "\tstart destroy suspend resume reboot shutdown undefin\n";
print "\tsmem setmem setvcpus\n"
print "\tclone migrate save restore\n";
print "\t't clone' needs the original (template VM name) to be added on -vm option using '-':\n";
print "\t'migrate' needs additional option:\n"
print "\t—destination <HOST>, where VMs are going to migrate:\n"
print "\t—live : used for live migration (default is paused)\n"
print "t undefine optional option '--remove' :\n"
print "\tused to delete their image file also\n"
print "\t'save' and 'restore' have optional option:\n"
print "\t—directory <VMs-nem-snapshot—path>\n"
print "\tcan also be assigned in '/etc/buna/buna.conf'\n"
print "\tdefault value is: '/var/lib/libvirt/store/ '<\n"
print "DEVICE—VALUES: are all mandatory values that 'virsh' command took except VM name\n"
print "\tattat—device [XML—of—the—device ]\n"
print "\tdetach—device [FILE]\n"
print "\tatat—disk [SOURCE-file—access—path] [TARGET—device—access—path]\n"
print "\tatat—interface [TYPE—network—or—bridge] [SOURCE—device]\n"
print "\tdetach—interface [TYPE—network—or—bridge]\n"
print "\tsee virsh manual for detail understanding\n";
1157 print "—connect <LIST–OF–HOSTS>: list of targeted hosts\n";
1158 print "\t—connect all: to apply on all hosts\n";
1159 print "\t\tdefault value is 'local–host qemu'\n";
1160 print "—vm <VM-name>: going to be managed, space is used as element separator\n";
1161 print "\t—vm [\{common1:template1 ... commonN\{subcommon1 ... subcommonM:template2 \{prefix1..10 postfix postfix1 postfix2 ... postfixN\} ...new–grouping–continue–here ...\}n\]
1162 print "\t\t—vm <all | active | inactive>: is used to refer all, working or not working VMs\n";
1163 print "\t\tuse '*' to present any substring\n";
1164 print "\t\t\tperiods (...) can be used for continuous numbering\n";
1165 }
1166 print "*** see buna documentation for detail understanding ***\n";
1167 print "*** see virsh and virt–clone ***\n";
This is the skeleton program that will be copied and modified based using rules file, when *buna* runs as a plugin.

### Listing A.4: skeleton.pl

```perl
#!/usr/bin/perl

## The Skeleton Program: that will going to be copied and modified by buna when it runs with option '--plugin <Daemon-Name> <Rules-File> & Rules-Block'. This program will not work unless Rules-Block is filled with If-Condition-Action codes, those are generated from Rules-File. buan Plugin-Process will do all modification on this file (not users). Users are allowed to write rules in any file using buan-rules-syntax feed to buna through the option stated above.

# Used packages
use Getopt::Long;
use List::MoreUtils qw{ any };
use List::MoreUtils qw / uniq /;
use Statistics::Descriptive;

# Global Variables
@head = ( 'count', 'running', 'blocked', 'paused', 'shutdown', 'shutoff', 'crashed', 'active', 'inactive', 'noofcpu', 'cusage', 'cpuusage', 'physicalmemory', 'guestmemory' );

# variables like host name storage, sample-size, time interval will be filled here
#Begin_Variable#

$SampleSize++;

# Blocked,Paused,Shutdown,Shutoff,Crashed,Active,Inactive,%CPU,Total memory (KB),Total guest memory (KB),Total CPU time (ns)
while (1) {
    @childs;
    $count=0;
    @data=();
    @statistics=();
    @lines=();
    @columens=();
    @statf=();

    foreach $host (keys $hostlist){
        $pid = fork();
        if ($pid) {
            # parent
```
$childs[$count] = $pid;

} elsif ($pid == 0) {

    # child areas

    $datafile = "$virttop" . "$host";

    $command = "$virttop --connect \"$hostlist\" $SampleSize $Interval --script --csv $datafile";

    system("$command");

    exit 0;

} else {

die "couldn’t fork e: $! \n";

}

$count++;

foreach (@childs) {

    $tmp = waitpid($_, 0);

}

foreach $host (keys %hostlist) {

    $datafile = "$virttop" . "$host";

    open(MYFILE, "$datafile") or die "couldn’t open the file \n";

    @lines = <MYFILE>;

    close(MYFILE);

    system("killall -9 virt-top");

    splice @lines, 0, 2;

    $i = 0;

    foreach $line (@lines) {

        @columens = split(' ', $line);

        $data{$host}{noofcpu}{$i} = $columens[3];

        $data{$host}{count}{$i} = $columens[4];

        $data{$host}{running}{$i} = $columens[5];

        $data{$host}{blocked}{$i} = $columens[6];

        $data{$host}{paused}{$i} = $columens[7];

        $data{$host}{'shutdown'}{$i} = $columens[8];

        $data{$host}{'shutoff'}{$i} = $columens[9];

        $data{$host}{crashed}{$i} = $columens[10];

        $data{$host}{active}{$i} = $columens[11];

        $data{$host}{inactive}{$i} = $columens[12];

        $data{$host}{cpuusage}{$i} = $columens[13];

        $data{$host}{physicalmemory}{$i} = $columens[14];

        $data{$host}{guestmemory}{$i} = $columens[16];

        $i++;

}

foreach $type (@head) {

    @statistics = analyze(@{$data{$host}{$type}});

    $statistics{$host}{$type}{mean} = $statf[0];

    $statistics{$host}{$type}{median} = $statf[1];

    $statistics{$host}{$type}{mode} = $statf[2];

    $statistics{$host}{$type}{percentile} = $statf[3];

    $statistics{$host}{$type}{range} = $statf[4];

    $statistics{$host}{$type}{min} = $statf[5];


$statistics{\$host}{\$type}\{max\}=\$statf[6];

if \{\$host\}{\$type}\{max\}=\$statf[6];

#Begin_Rule#

#End_Rule#

### Buna built in functions for Load balance service

sub LoadBalance{
  # ( host name 1 , host name 2 , memory/cpu/both , percent for tolerate ←
  # not to balance )

  $cpubalanced = 'no';
  $memorybalanced = 'no';
  $cs= $ms= $cd= $md= $ct= $mt= '';
  $sh1=$_[0];
  $sh2=$_[1];
  $balancetype=$_[2];
  $tolerate=$_[3];
  $n1= $statistics{$sh1}{active}{percentile};
  $n2= $statistics{$sh2}{active}{percentile};

  if ($balancetype eq 'cpu' or $balancetype eq 'all'){
    $c1= $statistics{$sh1}{cpuusage}{percentile};
    $c2= $statistics{$sh2}{cpuusage}{percentile};
    $tolerate=95 unless ($tolerate);
    if ($tolerate >= $c1 and $tolerate >= $c2){
      $cpubalanced= 'yes';
    }
  } else {
    $n1 = ($n1/(4 * $c1)) * ($c1 - $c2);
    $n1 = $n1 / .\.* /;
    $n2 = ($n2 / (4 * $c2)) * ($c2 - $c1);
    $n2 = $n2 / .\.* /;
    if ($n1 > 0){
      (\$cs,\$cd,\$ct) = ($sh1,\$sh2, $n1);
      if ($balancetype eq 'cpu'){
        $command="buna --action migrate --vmg active --amount←
        Set --source $hostlist{$cs} --destination ←
        $hostlist{$cd}";
        system("$command");
        return;
      }
    }

  } elsif ($n2 > 0){
    (\$cs,\$cd,\$ct) = ($sh2,\$sh1, $n2);
    if ($balancetype eq 'cpu'){
      $command="buna --action migrate --vmg active --amount←
      Set --source $hostlist{$cs} --destination ←
      $hostlist{$cd}";
      system("$command");
    }
  }
}

169
        return;
    } else {
        $cpubalanced='yes';
    }
}

if ($balancetype eq 'memory' or $balancetype eq 'all') {
    "$M1\$statistics\{h1\} \{physicalmemory\} \{percentile\}:
    "$M2\$statistics\{h2\} \{physicalmemory\} \{percentile\}:
    "$m1\$statistics\{h1\} \{guestmemory\} \{percentile\}:
    "$m2\$statistics\{h2\} \{guestmemory\} \{percentile\}:

    $m1=1 unless ($m1);
    $m2=1 unless ($m2);
    return unless ($M1 or $M2);
    $Rm1=($m1/$M1) * 100;
    $Rm2=($m2/$M2) * 100;

    if ($tolerate >= $Rm1 and $tolerate >= $Rm2) {
        $memorybalanced='yes';
    } else {
        $m1 = (($N1 + $M1)*(4 + $m1)) / (($m1/$M1) - ($m2/$M2));
        $m2 = (($N2 + $M2)*(4 + $m2)) / (($m2/$M1) - ($m1/$M1));
        if ($m1 > 0) {
            ($ms, $md, $mt) = ($h1, $h2, $m1);
            if ($balancetype eq 'memory') {
                $command= 'buna --action migrate --vmg active --amount=' $m1 --source $hostlist{$ms} --destination '$hostlist{$md}';
                system("$command");
                return;
            } else if ($m2 > 0) {
                ($ms, $md, $mt) = ($h2, $h1, $m2);
                if ($balancetype eq 'memory') {
                    $command= 'buna --action migrate --vmg active --amount=' $m2 --source $hostlist{$ms} --destination '$hostlist{$md}';
                    system("$command");
                    return;
                } else {
                    $memorybalanced='yes';
                }
            }
        } else {
            ("$cpubalanced -- $memorybalanced\n"
            "$Sh1 $Sh2 $balancetype $tolerate $N1 $N2 $m1 $m2 $M1 $M2 $Sm1 $Sm2 $Sm2\n";
        }
    }
}
print "all —— equal\n" and return if($cpubalanced eq 'yes' and $memorybalanced eq 'yes');

if($cpubalanced eq 'no' and $memorybalanced eq 'yes') {
    $command = " buna --action migrate --vmg active --amount $ct --source $hostlist{$cs} --destination $hostlist{$cd}";
    system("$command");
    return;
}

if($cpubalanced eq 'yes' and $memorybalanced eq 'no') {
    $command = " buna --action migrate --vmg active --amount $mt --source $hostlist{$ms} --destination $hostlist{$md}";
    system("$command");
    return;
}

if($cpubalanced eq 'no' and $memorybalanced eq 'no') {
    if ($cs eq $ms) {
        if ($ct <= $mt) {
            $command = " buna --action migrate --vmg active --amount $ct --source $hostlist{$cs} --destination $hostlist{$cd}";
            system("$command");
            return;
        } else {
            $command = " buna --action migrate --vmg active --amount $mt --source $hostlist{$ms} --destination $hostlist{$md}";
            system("$command");
            return;
        }
    } else {
        return;
    }
}

sub analyze {
    my @array = @_;  # print "@array\n";
    my $Percent = "$._[1]";
    my $stat = Statistics::Descriptive::Full->new();
    $stat->add_data(@array);
    my $mean = sprintf("%.2f", $stat->mean());
    my $median = sprintf("%.2f", $stat->median());
    my $mode = sprintf("%.2f", $stat->mode());
    my $percentile = sprintf("%.2f", $stat->percentile($Percent));
    my $min = sprintf("%.2f", $stat->min());
    my $max = sprintf("%.2f", $stat->max());
    my $range = sprintf("%.2f", $stat->sample_range());
    return ($mean, $median, $mode, $percentile, $range, $min, $max);
}