UNIVERSITY OF TARTU

FACULTY OF MATHEMATICS AND COMPUTER SCIENCE

Institute of Computer Science

Computer Science speciality

Mati Vait

Platform Virtualization for F2F Computing

Bachelor Thesis (6 EAP)

Supervisors: Ulrich Norbisrath, PhD

Artjom Lind, MSc

Author: ....”......” June 2011
Supervisor: ....”......” June 2011
Supervisor: ....”......” June 2011

Allowed to defence
Professor: ....”......” June 2011

TARTU 2011

Platform Virtualization for F2F Computing

Mati Vait

Contents

1 Background **6**

1.1 Benefits and drawbacks of virtualization 6

1.2 Available virtualization methods 7

1.2.1 Virtualization techniques 7

1.2.2 Virtualization technologies 9

1.3 Network virtualization 10

2 Architecture **12**

2.1 Requirements and examples 12

2.1.1 General requirements 12

2.1.2 Usage examples 14

2.2 Architecture of the solution 15

2.2.1 Components 15

2.2.2 Architectural layers 16

2.3 Ideas for Virtualized Platform Setup 17

2.3.1 F2F Computing Peer configuration 17

2.3.2 Virtual machine image provisioning 18

3 Prototype implementation **20**

3.1 Components 20

3.2 F2F Computing Job 21

3.2.1 The structure of F2F Computing job 22

3.2.2 Helper module and other tools 22

3.3 Networking in the virtual platform 23

3.4 Virtual machine configuration 23

4 User's Manual **25**
4.1 Dependencies . . . 25
4.2 Configuration . . . 26
4.3 Deployment . . . 27

Bibliography **30**

Introduction

The aim of this thesis is to research and provide a proof-of-concept solution for running distributed computing tasks and applications on a F2F Computing network using the Qemu virtual machines communicating over VDE (Virtual Distributed Ethernet)[21].

The use of virtual machines will make developers of distributed jobs independent from the underlying platform accessible to F2F Computing while providing them with more flexibility than the F2F Computing possibly could alone do. The virtual machine images can be prepared keeping in mind the specific task at hand. All of the required libraries and tools can be previously installed based on the application requirements and data for computation can be distributed in these virtual machines leaving the F2F Computing job with a little to do: launching a virtual machine and providing it with the network connection.

Additional resources could be assigned to the F2F computing cluster by connecting these virtual or physical machines over the VDE. For example, a high-performance computation cluster or storage server, which otherwise would be inaccessible in easily configurable manner from the regular F2F Computing network, can be included to the computational cloud by adding a network route to it on VDE.

Currently it has already been tried out by the F2F Computing developers, that it is possible to submit a F2F Computing slave instance as job to a Grid. Next the F2F Computing job was submitted to that F2F Computing slave node and successfully run on the Grid infrastructure utilizing the resources for given Grid node instance.

On the other hand, this work is a part of the exploration in F2F Computing for new directions. Various ideas are currently under consideration to improve the user experience and also the overall structure and the aims of project which will not be further discussed here.

Related work

In the field of personal spontaneous clouds there has not yet been much work done. A lot of commercial services and open source projects are available for building and running ordinary but heavyweight private clouds[1]. Most of these are intended for large scale users. The problem is that none of these provide a convenient platform for spontaneously setting up and running private personal clouds.

Amazon's EC2 (Elastic Cloud) [1] is a cloud computing service provided by Amazon. It enables to run virtual machines and use its storage (S3) on its infrastructure for a fee. Spontaneous setup for EC2 is not overly complex either through the web interface [2] or using the tools provided, but using it costs after certain amount of time[2].

OpenStack project [15] provides tools for setting up entire cloud infrastructure which is too much for simple testing and running small scale computations. OpenStack consists of three bigger classes of components: OpenStack Compute for starting, stopping and controlling virtual machine images, OpenStack Object Storage for storing massive amounts of various data (backups, applications, etc) and OpenStack Imaging Service for providing virtual machine images. Multiple services ([11]) are running on each computational node to support the cloud infrastructure; root access or sudo-enabled user account is required to install the software.

Condor is a HTC (High Throughput Computing) software [5] developed at University of Wisconsin-Madison. Computational node installation for Condor requires the root access [6] and adding a new user, modifications to the system to enable automatic startup and idle-monitoring. The installation can be done either manually or by using Perl install script. Condor node software runs in the background of the host machine and monitors for idling periods during which it can process the jobs.

Boinc is a software for grind computing [4] that is similar to the Condor. It also runs jobs during the idle periods and also requires root access installation which is intrusive [10].

From the viewpoint of a small scale user who only occasionally needs considerably more computational power, current solutions for distributed computation

[1] Amazon's EC2, Eucalyptus, OpenStack, etc.

[2] At the time of writing this thesis the Amazon is offering a certain amount of free service for new sign ups [3].

are cumbersome to set up and to run. On one hand, the cloud setups are either costly or too heavyweight to set up and on the other hand the grid middleware is designed to run job written in a specific manner. The installation process for both categories is intrusive and technical which makes recruiting friends and colleagues to provide computational power of their computers difficult if it is hard for them to set up the system. These reasons make both categories of distributed platforms unsuitable as a base for spontaneous private clouds.

Chapter 1

Background

1.1 Benefits and drawbacks of virtualization

Virtualization provides multiple benefits for running distributed applications over deploying them directly on the unvirtualized host system. First of all, virtualization reduces significantly the requirements for the host system. Host systems configuration can be oriented more towards running multiple virtual machines simultaneously, rather than to satisfy various library requirements that build up rapidly when system has multiple services running. Recovery from crashes for virtual machines means booting up the last backup of the image, but recovery for the crashed unvirtualized system means usually rebuilding it from the scratch. Host systems can vary in architecture and can be running different operating systems as long as the virtualizing application supports those and is able to run given virtual machine images.

Virtualization provides sandboxing for the deployable applications by separating them from the underlying environment. Sandboxing protects the host machine from potentially malicious applications from corrupting or stealing the data, installing unwanted software, crashing the system or using it as a proxy for launching new attacks against other systems.

On the downside, virtualization introduces additional computational overhead that consists of additional work done by the processor and also the additional memory that is used to keep each virtual machine instance running. The amount of additional work varies depending on the level of virtualization. The rule of thumb is that the lower level the virtualization takes place, the less

overhead occurs. That is because of the smaller number of abstraction layers involved.

In the next section I look more closely to some of the main virtualization techniques that could be used to solve the problem at hand: platform virtualization for F2F Computing.

1.2 Available virtualization methods

In this section I introduce and compare different virtualization approaches in order to choose suitable method for doing the platform virtualization for F2F Computing. How to virtualize, more specifically what tools to use for platform virtualization, is a big question since the virtualization can be performed on multiple different levels. Each of those levels have some positive and some negative sides to them, so it is necessary to understand these.

1.2.1 Virtualization techniques

Chroot or jailed environment can be considered as a basic idea of virtualization. It is done on the application level in the host system by setting up a separate root filesystem in a new directory. Applications, that are run under that new root folder, are given an impression that this filesystem is the only filesystem and as a result they know nothing about the actual filesystem on the host machine.

Although chroot is simple to set up for just application sandboxing it is not a good option to do platform virtualization for F2F Computing. Mainly because of it has no real encapsulation from the host's system. It can be damaged accidentally by the users of the host system. There is also a security risk involved due to the variety of ways of escaping [13] the chroot environment and possibility of altering maliciously the host system.

Application level virtualization is done for each application separately and each of those applications are separated from the others but are still running on the host system.

Virtualized application has its own address space and it sees the operating system as a generic operating system mostly having no idea if it is virtualized or not. Application level virtualization enables to deploy the same application on

similar kind of environments regardless if the system does or does not provide certain libraries because all of the dependencies come with the application.

Process-level virtualization is done by programming language specific interpretators. Two examples of this are Java Virtual Machine (JVM) and Python Virtual Machine(PVM). In case of process-level virtualization the code written in high-level language is translated into intermediate byte-code and then processed by the language interpretator.

Paravirtualization is a virtualization technique in which the guest operating system's kernel is modified to use modified CPU instructions so that the host CPU underneath can understand and run faster. The guest system uses modified drivers to connect directly to the host machine. The main benefit of paravirtualization is the performance. for an example it helps to acquire more precise time measurements[1] on the guest. The gain in performance is achieved mostly through the fact, that the guest system is aware of it's role (virtual machine) and uses more efficient system calls and modified device drivers to access system resources.

Full virtualization is a technique, in case of which an unmodified guest operating system is running on some host machine and is not aware of that it is a virtual machine. Every system call, that is made to the host machine from the virtualized operating system, is captured and executed by the machine emulator. Virtual machine uses unmodified drivers to access system resources.

Full virtualization can be divided into two categories: native virtualization and hosted virtualization. In the case of the native virtualization, the hypervisor[2] is located directly on top of the host hardware and is running without any help of the host operating system[3]. However the negative side to it is that in this mode it is hard to control the amount of resources used by the virtual machine.

In case of the hosted virtualization, the hypervisor is located between the host operating system and the guest operating system where the unsafe system calls are rewritten. This way the control over the virtual machine's and the host's performance is better. To run guest operating system's system calls

[1] In most of the applications having a precise time is a must.

[2] Hypervisor is an application that translates system calls between the host and guest operating system.

[3] Hypervisor may be directly built into the firmware of the host systems

natively full virtualization requires hardware support such as Intel's VT-x or AMD's AMD-V and a guest virtual machine built for the same architecture.

1.2.2 Virtualization technologies

Xen is an open source hypervisor[25] that is considered to be the industry standard. It provides an abstraction level between host hardware and multiple guest operating systems. Xen is designed to consolidate together a relatively large number of virtual machines on a single physical machine while not loosing much in performance.

Xen, as a hypervisor, is located in between of physical machine and the guest (virtual) machine. In addition to the guest machines, a Xen setup always has a *domain0* – an operating system that is always started first by the hypervisor and is used to launch other virtual machines. Xen uses paravirtualization technique between the hypervisor and the guests to provide uniform and secure access to the hardware resources. Most of the hardware access is provided through the domain0 virtual machine in a generalized manner: the underlying machine architecture is simplified and presented to all of the guests as a generic system.

Because of very intrusive installation process[18], which involves setting up specially modified operating system - *domain0*, from where guest machines could be launched, Xen is not the best option for running a regular node's virtual machines in F2F Computing network. Xen installation requires changing the host's kernel or even reinstalling the whole system. Xen suits better for running special-purposed nodes that are not restarted or reconfigured very often on F2F Computing network.

VMware is a company providing virtualization solutions, among which is a free product called VMware Player. It enables to create and use run machines on a workstation. VMware Player uses paravirtualization techniques to support wide range of guest operating systems.

Qemu [17] is a processor emulator that enables to run virtual machines directly in the userspace using full virtualization. Guest operating systems are run unmodified while Qemu provides a generic interface to the hardware. Qemu has built in support for many different kind of processor architectures and knows how to interpret their instructions.

KVM (Kernel-based Virtual Machine) [12] is an open source hypervisor that is used mostly in conjunction with Qemu. KVM is loaded as a kernel module and thus provides better performance than having Qemu running alone. KVM is built to take an advantage of a regular Linux kernel to run virtual machines in hosted mode, so it does not have to do much of the hardware controlling itself. Because of the reuse of the Linux kernel, the KVM hypervisor is located closer to the hardware than Xen.

UML-Linux (User Mode Linux) [19] is virtualization solution in which Linux kernel is started and run in the user-space and all calls from and to the guest system are intercepted and rewritten as necessary, by UML-Linux. Complete virtual machines are built using UML-Linux by attaching additional resources, such as filesystems, network interfaces and other devices, to the kernel. All virtual machines are run under user privileges and without any direct low level access to the host machine hardware. This means that all virtual machines run in this manner are quite slow and probably not suitable to be used for platform virtualization in general.

VirtualBox [23] is a free and relatively popular virtualization solution that is developed by Oracle. It uses hosted mode virtualization and supports wide range of guest operating systems.

1.3 Network virtualization

Network virtualization is the process of creating subnets that span over multiple real or virtual subnets. Virtual networks may consist both of guest and host machines. Multiple techniques exist that can be used to create virtual networks but mainly they are formed by creating network interfaces to the guests and then routing the connections between these interfaces using some means for transport. In the virtualized networks the data throughput is reduced because of the overhead of encapsulation of the traffic into the lower levels of network.

Virtual networks are made utilizing various tools and technologies. In the remaining part of this section I give an overview of some more useful of these.

Network Address Translation (NAT) is used to seamlessly manipulate the traffic coming from one subnet and going to another subnet to look like it is actually from that second subnet. The purpose of NAT is to hide one or multiple

private subnetworks behind one IP address (the gateway). NAT was introduced because of IPv4 shortage. Perfect to hide internal hosts from external attacks, however it also complicates P2P communication.

Virtual Private Network more ofter referred to as VPN is a secure way of building Private Networks on top of the existing IP networks. All of the traffic is protected by encryption. VPN encapsulates multiple service channels into one channel. This encapsulation reduces the router load by mapping connections over the tunnel into respective ports on the VPN instances on either side of the tunnel.

Virtual Distributed Ethernet (VDE) is a subproject of Virtual Square project. VDE provides a fundamental toolkit for building virtual network solutions. VDE toolkit consists of software-based switch and wide range of utilities for connecting switches to other switches, network interfaces to switches and other endpoints such as virtual machines[4]. Spanning tree protocol is implemented on the vde_switch tool and it enables to route the traffic faster.

UML-Linux utilities [19] provide also means for network setup (uml_switch). They contain relatively big portion of the code from VDE project and may be considered to be used for extending the virtualized platform by UML-Linux virtual machines. Currently UML-Linux tools are not as advanced as VDE and thus are not used for this work.

Virtualbricks [24] project is based on KVM and Qemulator projects. It enables to graphically configure virtual network connections and virtual machines. This, and similar tools are worth looking into when platform virtualization is already more advanced for F2F Computing. There exists a potential of integrating similar tool into special F2F Computing job, so the the private cloud management could be more flexible and more under the control of the initiator.

[4]Qemu has built in support for VDE.

Chapter 2

Architecture

In this chapter I bring out the requirements along with some usage examples that drive the design and describe the overall architecture of the prototype. The main objective is to identify necessary components and specify how exactly the platform virtualization can be provided on the F2F Computing framework at the moment.

The requirements are brought out from the point of view that regular computational nodes are hosted on regular computers. These nodes may not have there full priorities over the host system resources.

2.1 Requirements and examples

2.1.1 General requirements

Virtualized computation environment must not depend on underlying host system. The virtualized computational environment must not have any other external dependencies apart from network connection and sufficiently reserved amount of memory and CPU resources.

Indistinguishable virtualized and physical platforms. All applications that run on one platform must also be able to run on the host platform. This requirement can be relaxed in a sense that not all special software that runs on specialized hardware can be run in virtualized settings and vice versa. However having this requirement fulfilled, means that development time can be significantly reduced while applications are built on physical platforms and deployed

on the virtual platform.

Minimal requirements on non-dedicated host system. Virtual machine must not exhaust too much resources on host machine and by that render it unusable through the host operating system. On the host systems that are used exclusively for running virtual machines or are purposely used for running virtual machines this requirement does not fully apply.

Use of high performance virtualization methods. Virtualization methods must be chosen such that they do not make virtual machines too slow compared to the physical machines. Capabilities of the average host machine must be considered when assembling a generic virtual machine for the majority of computational nodes.

Nodes in the virtualized platform must be interconnected in virtual network. The virtualized network runs over existing network connections. Having a separate logical layer of network addressing and traffic helps administrators of the network effectively group all of the components in platform. Using virtual network (see subsection 1.3) makes all of the remotely located components and communication between those components easily manageable.

Some of the components in the virtual network may also have external connections. These connections are necessary for managing, monitoring and extending the network.

Network connections in virtual network must be as optimal as possible. The optimality in this case means two things. Firstly it means that the transmission speeds in the virtual and in the underlying physical network should not differ significantly. Naturally there exists an overhead that occurs nevertheless when having virtualized network traffic encapsulated inside of the real network traffic. This overhead can be minimized by not artificially increasing the unnecessary workload for an example by running multiple ssh sessions one inside of another.

Secondly the optimality means that all of the communication happening in the virtual network is also routed via the fastest route possible. Fast routing can be achieved in two ways: either by adding optimal routes manually into the routing tables of virtual machines or by using automated and adaptive solutions such as fast spanning tree protocol implementation in software based switches.

Virtualized platform must be easy to set up. Depending on the role of the component in the platform it has to be as easy to set up as possible. Simple computational nodes must be launched using only minimal effort[1] on the user's part and must not require any special knowledge either on programming or system administration. Only if this requirement is fulfilled then is it possible to have as many regular users as possible for deploying the main force of the platform: generic computational nodes.

Virtualized platform must be easily extendable. The proper selection and the use of networking tools will enable the administrators to extend virtualized platform whenever the need or opportunity arises. For an example by adding another group of computational nodes to existing set of nodes through connecting some virtual network switches either directly or through tunnels.

2.1.2 Usage examples

Running a specific distributed computational job. A small group of virtual machines[2] are set up for development and testing of distributed computational jobs. Those machines are configured according to the needs of the application and are used as base images for creating virtual machines for all of the nodes.

F2F Computing job is developed keeping in mind specific needs of the distributed computational job. Required resources are negotiated over the instant messaging service, group chat is formed and F2F Computing job is distributed. Each F2F Computing job, based on the machine that it is running on, establishes initial network connections, configures and deploys it's virtual machine. During the boot time virtual machines are connected together into a virtual network and the actual distributed job or application is launched.

Adding additional resources to another F2F Computing cluster. For multiple possible usage scenarios it is necessary to augment existing computational network with additional resources. This can be done in multiple ways,

[1] Simple computational nodes are meant to be run on the machines operated by an average computer user who simply launches the virtual machine through running F2F Computing job and after that has no further interaction with the node.

[2] For an example base images for virtual machines of master and slave roles.

but if using F2F Computing to launch additional set of machines, the basic setup is similar to the previous paragraph up to the point where it is time to launch the computational job. In that point, instead of launching that job, the network is connected to the other network through some (or multiple) of the nodes in the newly created network that acts as a gateway.

The augmentation becomes necessary, when some of the computational nodes fail or network connections go down and remaining nodes cannot recover. More resources and redundancy can be added to the F2F computing cluster by adding additional friends and allowing them to add their friends. To build up more redundant system, some of the new nodes could be instead of launching their virtual machines back up some of the more critical ones already running.

2.2 Architecture of the solution

The architecture consists of multiple components of which some are concrete and some are logical entities by nature. In the following I discuss these components and then separate them into distinct levels. Some of the terminology is borrowed from the VDE project, because of the prototype solution makes a heavy use of tools that it provides.

2.2.1 Components

Host system is an underlying component of the virtualized platform. Hosts provide computational and storage resources for the virtualized platform. Host system is connected to other host systems and they form the basis for the virtualized platform. At this point it is not necessary to make a distinction between dedicated host[3] and regular host[4].

Guest system is a virtual machine that is the main component of the virtual platform. It is build for specific role depending on the overall task and carries libraries and other necessary tools. Sometimes it is justifiable for the data to be included also in the virtual machine[5].

[3] Dedicated host is a physical machine, that is used solely for providing some specific service or resource, for an example database.

[4] For a regular host only some of it's resources are allocated for the guest system.

[5] Whether the data is included in the virtual machine during the initial virtual machine distribution or not, depends on the sizes of the virtual machine and the data. Also the network speed must be taken into account: maybe it is more efficient to do the delivery by some other means.

Software-based switch is a component that is responsible for interconnecting different parts of the network. For the prototype solution there is always one switch placed in front of each of the virtual machines. This setup enables to use uniform configuration for all of the virtual machines in the initial F2F Computing job. Also, having a switch in front of the virtual machine provides an easy opportunity for attaching another network segment through this switch.

Wire is a logical component by nature because is comprises of several other applications. It is used to interconnect parts of either the underlying network or the virtual network, hence the name. Wires are used to make two types of connections: guest system to switch, switch to switch.

In the architecture of the solution it is possible to separate two basic levels: physical and virtual. All of the host systems are situated on the physical level. They may be located far apart in respect to each other and can be connected by different means such as ssh, VPN, LAN, P2P (TCP or UDP) or other.

2.2.2 Architectural layers

Components of the solution can be divided into three layers. Each of those layers can be considered separately depending on the components that given layer focuses on and the problems that the given abstraction helps to solve.

Virtual layer provides a high-level view for the developer who is writing applications for the virtualized platform.

This layer consists of virtual machines that are used to run either final distributed job or an application. All of the machines are provided implicitly with network connections so that they all appear to be on the same network. Figure 2.1 shows one possible virtual layer that has multiple virtual machine instances not taking into account whether they are running on the same host, same site or even if their hosts are connected to the same network segment. The only concern in virtual level is the configuration of the applications and implicitly provided network connections which are not shown.

Logical layer describes the topological setup of the virtual machine instances, how they all are connected to each other. Figure 2.2 shows two possible ways of setting up topologies (star and mesh), however it is possible to combine these

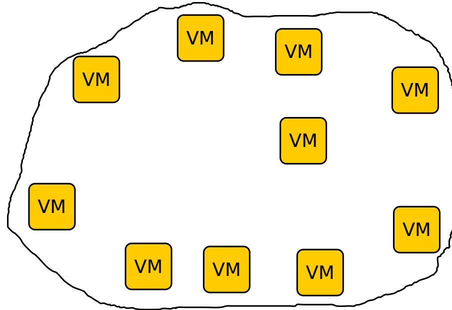

Figure 2.1: Layout of the components in the virtual layer.

and some other different network layouts to achieve close to optimal network performance.

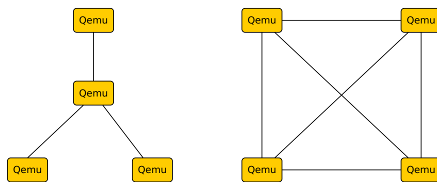

Figure 2.2: Star and Mesh layout of the components in the logical layer.

Physical layer is the most important layer from the viewpoint of this work. This layer concerns with the layout of different components for setting up the underlying infrastructure. Each line in the Figure 2.3 denotes a connection between two given components whereas that connection may not necessarily be a network connection[6].

2.3 Ideas for Virtualized Platform Setup

2.3.1 F2F Computing Peer configuration

The F2F Computing job is the glue for setting up and holding together the virtualized platform. It is essentially a regular F2F Computing job, which instead

[6]Connections between F2F Computing jobs and virtual machines that are launched from these jobs may also be implemented through local pipes.

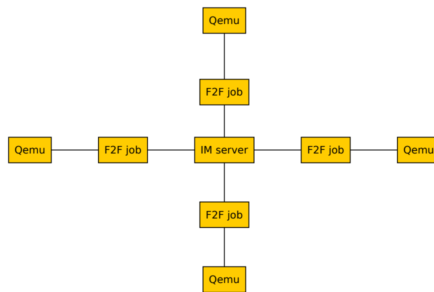

Figure 2.3: Layout of the components in the physical layer.

of performing calculations and returning results bootstraps the virtualized platform. It starts up virtual machines and establishes underlying connections for virtual network.

As in any distributed job that has separate master and slave roles, the F2F Computing job contains two parts. The master part of the F2F Computing job is for setting up master virtual machine and attaching it to the switch. Master job is also responsible for distributing parameters that need to be coherent across the network. These parameters are for an example the MAC addresses for all guest virtual machine network interfaces, special parameters for VDE switches.

Slave part of the F2F Computing job is for configuring and starting up the slave virtual machine and attaching it to the switch. When launched, the slave part receives parameters from the master job and configures it's virtual machine and prepares network connections.

2.3.2 Virtual machine image provisioning

Distribution of the virtual machine images can be done in multiple ways. The easiest way would be to retrieve the images by their URLs from dedicated server or to use some of data retrieval tools.

Second approach is to get the images directly from the master node or from special node in the F2F Computing network using the F2F Computing communication channels. In this case the speed and efficiency would all depend on the speed of the F2F Computing messaging channel.

Third option is to use F2F Computing messaging layer to set up TCP streams between the peers and share large files from master to slaves using swarm download algorithm where the master broadcasts the data to the slaves simultane-

ously.

The used distribution strategy also depends on what kind of virtual machine images are used on what kind of nodes and what are their roles. Some or all of the nodes may take part in the distribution process using Peer-to-Peer data transfer techniques to share uniformly over the network the same kind of virtual machine images[7].

Management. For every role in the virtual platform a virtual machine base image is created. That image is delivered to appropriate node (see previous paragraph) and the node uses that image to derive a working copy of it. The base virtual machine image is stored on the host's hard drive as is and it is used to derive a temporary virtual machine image that is actually launched[8].

[7] This approach is particularly fast when the number of distinct base images is small compared to the number of the nodes in the network.

[8] This approach enables clean restart for the guest and also creates an opportunity to later investigate virtual machine state if something went wrong.

Chapter 3

Prototype implementation

In this section I describe the F2F Computing job related code in more technical detail. Currently the prototype depends a lot on some of the common tools found in the Linux systems e.g. nc [14] due to the fact that it is a versatile tool with very low overhead that can be used to connect Qemu machines to F2F Computing job.

Python [16] was chosen as the implementation language because F2F Computing framework supports it and the code is written in Python with portability in mind.

3.1 Components

Following components and applications are used in the prototype solution.

F2F (Friend-to-Friend) Computing is a framework [7] which provides a platform for distributed computing borrowing ideas from cloud computing, Peer-to-Peer and social networks. It is a middleware running on top of instant messaging (IM) clients and provides means for writing spontaneously deployable distributed applications and services. Currently supported languages are python and C/C++. In the old version Java was supported and that is currently being redesigned and rewritten to be included in the future.

Openfire is an IM server, by Jive Software [9], that uses XMPP instant messaging [20] protocol. It enables users to have group chats - the feature that F2F Computing takes extensively advantage of.

Qemu is a processor emulator [17] which is capable of running virtual machine images of multiple different architectures on host machine with nearly native speeds when using kernel-based acceleration through KVM kernel module. In this thesis Qemu processor emulator is used to run virtual machines because it emulates a wide range of CPU architectures and is extensively configurable.

Debian GNU/Linux distribution is used as a operating system in the virtual machines. Other operating systems could also be used.

VDE tools. VDE (Virtual Distributed Ethernet) [21] is a sub-project of the Virtual Square [22] project that enables to create sparse virtual machines spanning over multiple real machines. VDE project provides a set of tools to establish networks containing virtual and real machines. The most basic set of tools, that enable to set up virtual network between two or more machines are vde_switch[1], vde_plug[2] and dpipe[3].

The central component of the VDE network is vde_switch. It is a software-based Ethernet-compliant network switch that supports fast-spanning-tree protocol. dpipe is a utility to connect vde_switch to other switches through other transport programs[4].

nc (netcat) is a versatile networking [14] tool which is used in the solution as a means for transport between F2F Computing job and vde_switch but there are other alternatives, such as ssh or to programmatically read and write the standard input and output, that could be used instead.

3.2 F2F Computing Job

The F2F Computing job is used to launch, configure and interconnect separately located virtual machines into a virtualized platform. The F2F Computing client is launched on each of the nodes and once the master node is launched, then the F2F Computing job is distributed among all of the clients. At this stage

[1] vde_switch is a software-based switch that is Ethernet compliant and implents fast spanning tree protocol for efficient routing.

[2] vde_plug is an endpoint of a logical "wire" connecting two vde_switches or a vde_switch and a virtual machine.

[3] dpipe is a bidirectional pipe command, that connects standard output of one process to the standard output of another process.

[4] In this work nc and F2F Computing messaging was used for connections, however network sockets or other means could be used.

all of the nodes in the group set up their virtual machines and interconnect them into VDE network. All of the communication is proxied through the F2F Computing messaging layer. The only external component needed to have a working communication between nodes is a plain XMPP server (Openfire, see section 3.1). That server is also used for managing all F2F Computing users in the network.

The F2F Computing job code of the prototype is divided into two main parts: sample network topologies (see 2.2.2) and a helper module for setting up and configuring virtual machines and networking.

3.2.1 The structure of F2F Computing job

A F2F Computing job, that sets up virtual platform, consists of two parts: master and slave role which in general can be referred as master and slave nodes. Both of them share common logic for setting up network traffic forwarding from vde_switch to appropriate F2F Computing job on the other side of the network where it is in turn forwarded into vde_switch.

Master node firstly generates parameters and configures its virtual machine. The virtual machine is started in parallel to the master node as soon as it is configured. Master virtual machine is launched first to ensure, that all nodes can receive a network address through DHCP. After starting the virtual machine, the master node generates and provides parameters for F2F Computing nodes to configure their virtual machines. All of the unconfigured slave nodes are provided with parameters and at this point the master node continues to wait for added new nodes and provide these with configuration parameters.

Slave node receives parameters from the master node and launches the virtual machine also in parallel. While the virtual machine is starting up, the slave node prepares traffic forwarding to either to the center node of the star-topology or to every other node in the mesh-topology. It depends on which of the proof-of-concept implementations is currently running.

3.2.2 Helper module and other tools

For performing repetitive tasks and to reduce boilerplate code, a great part of the code was moved into a separate module. It contains functionality for easily configuring and launching virtual machines and connecting these machines to

the VDE network. This Python module is also separately usable from the F2F Computing job.

3.3 Networking in the virtual platform

Virtual network configuration for the virtualized platform can be done in two ways. First option is to have static addresses which means that each machine on the network must be individually preconfigured and possibly shipped while containing this information. The second, and more preferable, option is to use dynamic addressing which divides all virtual machines in the network into two classes: master and slaves. Master machines have DHCP[5] servers running and provide slave virtual machines with IP addresses and possibly some other network related configuration information.

Different topologies. Two different network topologies (Figure2.2) were implemented as a result of this thesis. Both of the implementations as F2F Computing jobs can be found on the DVD attached to the thesis.

- f2f_STAR.py - Star topology implementation,
- f2f_MESH.py- Mesh topology implementation.

Software-based switches (vde_switch) are located in front of each of the Qemu virtual machines to enable to change the topologies more easily. Otherwise if only some central switches were used, then it would be difficult to find a suitable point to connect additional machines and networks.

Another motivation for having such high concentration of switches is to use the fast spanning tree protocol in vde_switches to achieve the most effective routing possible.

3.4 Virtual machine configuration

Depending on the role in the virtualized platform, the virtual machines are configured differently. For the proof-of-concept solution following simple configuration was set up.

[5]DHCP - Dynamic Host Configuration Protocol

Master virtual machine is configured with a static IP address and is providing a DHCP service.

Slave virtual machines are configured to automatically acquire the IP address over the DHCP protocol. All slaves also have preinstalled ssh public keys of the master virtual machine to lessen the access and configuration complexity.

Testing, diagnostics. To ensure the health and responsiveness of the platform, each of the virtual machines should to be examined separately, albeit often it is sufficient to examine the state of the network from one or two nodes to verify that all other nodes are up and responding. For that purpose each virtual machine has installed tcpdump and nmap applications.

Figure 3.1 shows an example of how to verify that all of the (four) launched nodes are actually running and responsive. Firstly, the IP of the local machine is queried and secondly, all 256 IP addresses are pinged in given subnet of 192.168.254.0/24.

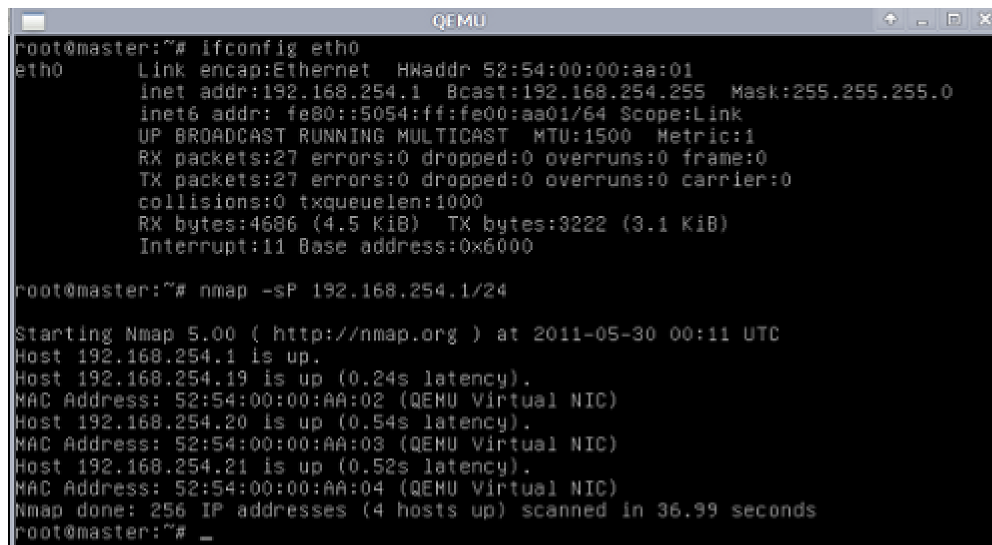

Figure 3.1: Using ifconfig to find out the IP of the network interface and nmap to display all of the machines in the subnet.

Chapter 4

User's Manual

In this section I describe concrete dependencies, configuration and deployment of the prototype solution for a Debian Linux based system.

On Debian Linux the Qemu package already has a built in support for VDE. On Ubuntu, since version 10.4 and later, the support for VDE is not included due to some maintenance issues. For Ubuntu the Qemu package has to be recompiled and reinstalled to enable VDE network interfaces.

4.1 Dependencies

In this section I list dependencies for running the prototype code on host machines.

1. Openfire server. Download a Linux version (tar.gz) of the Openfire server from
   http://www.igniterealtime.org/downloads/index.jsp#openfire. Only one instance of this server is needed to set up. Unpack the archive using some graphical archiver or by issuing command:

   ```
   tar zxvf openfire*.tar.gz
   ```

2. Qemu package. Install the package on Debian based systems by issuing command as a root:

   ```
   apt-get install qemu qemu-kvm
   ```

3. Virtual machines for slave and master nodes can be taken from the DVD media (respectively slave.qcow2 and master.qcow2).

4. VDE tools are available under the name vde2 in the Debian repositories. Issue following commands a root to install:

   ```
   apt-get install vde2
   ```

5. F2F Computing core and python bindings are necessary to run the F2F Computing nodes. Latest version of the source for the core can be downloaded from [8]. This page contains extensive instructions for setting up and building the core. An alternative option is to take pre-built 32bit binaries from the DVD media.

6. F2F Computing headless slave nodes can be taken from the DVD media (slave*.py).

7. F2F Computing headless master nodes can be taken from the DVD media (master*.py).

4.2 Configuration

Following steps are necessary to prepare environment for launching. The configuration of the virtual machines in step 5 can be skipped because sample virtual machines are already configured and can be used instead.

1. Openfire

   (a) Start the server

   ```
   openfire/bin/openfire start
   ```

   (b) Disable offline messaging:

   i. configure administrator password and log into the server,

   ii. in Server→Server Manager→Server Settings→Offline Messages enable "Drop",

   iii. save the settings.

   (c) Create users f2f01, f2f02, f2f03, f2f04,... (password e.g. f2f):

   i. log in as administrator,

ii. in Users/Groups→Create New User enter the username of the new used and according password,

iii. click Create User.

2. Build the F2F Computing core or take pre-built 32bit binaries from the accompanied DVD. Put binaries and bindings to the same folder as the slave and master node files are (for testing on a single machine).

3. Set user parameters in F2F Computing nodes. Edit the computing nodes slave*.py and master*.py to contain previously, in Openfire server, configured usernames and passwords.

4. Set the location for virtual machine images in the mesh/star F2F Computing job file.

5. *Configure virtual machines

   (a) not to remember MAC and IP address associations (udev).

   (b) to ask for an IP over DHCP (slaves)

   (c) to provide IP addresses over DHCP (master)

4.3 Deployment

1. Start Openfire server

```
openfire/bin/openfire start
```

2. Launch slave nodes

```
python slave1.py
```

```
python slave2.py
```

```
python slave3.py
```

3. Launch master node

```
python master_MESH.py # mesh topology or
python master_STAR.py # star topology
```

Summary

The main contribution of this thesis is the prototype solution of platform virtualization for F2F Computing. I provide two possible network topologies of setting up the virtualized platform: star and mesh. Both are implemented as F2F Computing jobs using some of the common code from a helper module I wrote for launching Qemu virtual machines.

In the thesis I also look into various virtualization techniques and technologies. Some of them could be used in the virtual platform for some persistent tasks. For an example the Xen hypervisor could be used instead of the Qemu for running database or storage servers that are less changing parts of the virtualized platform.

One possible future research direction after implementing platform virtualization for F2F Computing could be the integration of graphical network and virtual machine manager software like Virtualbricks[1] to F2F Computing platform. This would enable to central configuration for all of the networked machines from the master node using graphical interface.

[1] http://virtualbricks.eu/

Platvormi virtualiseerimine F2F Computing raamistikule

Mati Vait
Bakalaureusetöö (6 EAP)
Sisukokkuvõte

F2F Computing raamistikku on arendatud Tartu Ülikoolis juba pikemat aega ning käesolevaks hetkeks on see juba võrdlemisi paindlik, sest jooksutab C/C++ keeltes kirjutatud hajustöid LLVM (Low Level Virtual Machine) virtuaalmasina kaudu ja toetab ka Pythonis kirjutatud töid.

Käesoleva töö eesmärk on uurida virtuaalmasinate jooksutamist F2F Computing arvutusvõrgus sihiga see virtualiseerida.

Arvutusplatvormi täielikul virtualiseerimisel on mitmeid eeliseid: spontaansus ülesseadmisel, suurem valik kasutatavat tarkvara mida kasutada hajustööde loomiseks ja jooksutamiseks, võimalus virtuaalmasinaga kaasa panna arvutuste algandmed. Lisaks tavapärastele hajustöödele on võimalik virtualiseeritud platvormil käivitada hajusrakendusi, näiteks mitmest komponendist koosnevaid veebiteenuseid. Muuhulgas on hõlpsalt võimalik laiendada virtuaalset platvormi väliste ressurssidega. Laiendamine toimub ühenduse lisamisega arvutusvõrgust ressursini.

Käesoleva töö raames uurisin erinevaid virtualiseerimise tehnikaid ning vahendeid. Vaatasin kuidas võiks välja paista virtualiseeritud platvorm erinevate kasutajate vaatepunktist ning praktilise osana valmis prototüüplahendus F2F Computing arvutusplatvormi virtualiseerimiseks. Lahenduses on realiseeritud kaks topoloogiat (Star ja Mesh).

Bibliography

[1] Amazon Elastic Compute Cloud (Amazon EC2). http://aws.amazon.com/ec2/, May 2011.

[2] Amazon Elastic Compute Cloud: Get Started with EC2. http://docs.amazonwebservices.com/AWSEC2/2011-05-15/GettingStartedGuide/, May 2011.

[3] Amazon Web Services (AWS) free usage tier. http://aws.amazon.com/free/, May 2011.

[4] BOINC - Open-source software for volunteer computing and grid computing. http://boinc.berkeley.edu/, May 2011.

[5] Condor project homepage. http://www.cs.wisc.edu/condor/, May 2011.

[6] Condor version 7.6.0 manual: 3.2 installation. http://www.cs.wisc.edu/condor/manual/v7.6/3_2Installation.html, May 2011.

[7] F2F - ulno.net. http://ulno.net/f2f/, June 2011.

[8] F2F Development - ulno.net. http://ulno.net/f2f/development/, June 2011.

[9] Ignite Realtime: Openfire Server. http://www.igniterealtime.org/projects/openfire/, May 2011.

[10] Installing BOINC - BOINC. http://boinc.berkeley.edu/wiki/Installing_BOINC, May 2011.

[11] Installing OpenStack Compute on Ubuntu. http://docs.openstack.org/bexar/openstack-compute/admin/content/ch03s02.html, May 2011.

[12] KVM - main page. http://www.linux-kvm.org/page/Main_Page, May 2011.

[13] [Linux University Handbook]: chroot. http://wiki.linux.edu/chroot, May 2011.

[14] Manual Pages: nc. http://www.openbsd.org/cgi-bin/man.cgi?query=nc, May 2011.

[15] OpenStack projects. http://www.openstack.org/projects/, May 2011.

[16] Python Programming Language - Official Website. http://www.python.org/, May 2011.

[17] *Qemu - open source processor emulator.* May 2011.

[18] RHEL6Xen4Tutorial: Xen installation - Xen Wiki. http://wiki.xensource.com/xenwiki/RHEL6Xen4Tutorial, May 2011.

[19] The User-mode Linux Kernel Home Page. http://user-mode-linux.sourceforge.net/, May 2011.

[20] The XMPP Standards Foundation. http://xmpp.org/, May 2011.

[21] VDE - Virtual Distributed Ethernet. http://vde.sourceforge.net/, May 2011.

[22] Virtual Square. http://www.virtualsquare.org/, May 2011.

[23] VirtualBox. http://www.virtualbox.org/, May 2011.

[24] Virtualbricks. http://virtualbricks.eu/, May 2011.

[25] Xen® hypervisor. http://xen.org/, May 2011.