Aalto University
School of Science
Degree Programme of Computer Science and Engineering

Risto Laurikainen

Improving the efficiency of deploying virtual machines in a cloud environment

Master's Thesis
Espoo, January 16, 2012

Supervisor: Professor Jukka K. Nurminen
Instructor: Pekka Lehtovuori, Ph.D., CSC

Aalto University
School of Science
Degree Programme of Computer Science and Engineering

ABSTRACT OF
MASTER'S THESIS

Author: Risto Laurikainen

Title:
Improving the efficiency of deploying virtual machines in a cloud environment

Date: January 16, 2012 **Pages:** 88

Professorship: Data Communication Software **Code:** T-110

Supervisor: Professor Jukka K. Nurminen

Instructor: Pekka Lehtovuori, Ph.D.

Flexible allocation of resources is one of the main benefits of cloud computing. Virtualization is used to achieve this flexibility: one or more virtual machines run on a single physical machine. These virtual machines can be deployed and destroyed as needed. One obstacle to flexibility in current cloud systems is that deploying multiple virtual machines simultaneously on multiple physical machines is slow due to the inefficient usage of available resources.

We implemented and evaluated three methods of transferring virtual machine images for the OpenNebula cloud middleware. One of the implementations was based on BitTorrent and the other two were based on multicast. Our evaluation results showed that the implemented methods were significantly more scalable than the default methods available in OpenNebula when tens of virtual machines were deployed simultaneously. However, the implemented methods were slightly slower than the old ones for deploying only one or a few virtual machines at a time due to overhead related to managing the transfer process.

We also evaluated the performance of different virtual machine disk formats, as this choice also affects the deployment time of the machine. Raw images, QCOW2 images and logical volumes were evaluated. Logical volumes were fastest overall in sequential disk I/O performance. With sequential reads and writes, raw images could provide at best approximately 88% of the write performance and 95% of the read performance of logical volumes. The corresponding numbers for QCOW2 were 86% write and 74% read performance. Random access performance between QCOW2 and raw images was nearly identical, but LVM random access performance in our specific benchmark was significantly worse.

If the usage pattern of the cloud is such that deploying large batches of virtual machines at once is common, using the new transfer methods will significantly speed up the deployment process and reduce its resource usage. The disk access method should be chosen based on what provides acceptable performance for the task being executed and provides the fastest deployment times.

Keywords: cloud environment, cloud computing, BitTorrent, multicast, OpenNebula, virtual disk, virtual disk formats

Language: English

Aalto-yliopisto
Perustieteiden korkeakoulu
Tietotekniikan tutkinto-ohjelma

DIPLOMITYÖN
TIIVISTELMÄ

Tekijä: Risto Laurikainen

Työn nimi:
Virtuaalikoneiden käyttöönoton tehostaminen pilviympäristössä

Päiväys:	16. tammikuuta 2012	**Sivumäärä:**	88
Professuuri:	Tietoliikenneohjelmistot	**Koodi:**	T-110
Valvoja:	Professori Jukka K. Nurminen		
Ohjaaja:	FT Pekka Lehtovuori		

Resurssien joustava jakaminen on yksi pilvilaskennan merkittävimmistä eduista. Virtualisointia käytetään tämän joustavuuden saavuttamiseksi: yhtä tai useampaa virtuaalikonetta ajetaan yhdellä fyysisellä koneella. Näitä virtuaalikoneita voidaan ottaa käyttöön ja tuhota tarpeen mukaan. Yksi este joustavuudelle nykyisissä pilvijärjestelmissä on usean virtuaalikoneen samanaikaisen käyttöönoton hitaus, joka johtuu tähän käytettävissä olevien resurssien tehottomasta käytöstä.

Toteutimme ja vertailimme OpenNebula-väliohjelmistolle kolme eri menetelmää siirtää virtuaalikoneiden levykuvia. Yksi menetelmistä perustui BitTorrentiin ja kaksi muuta multicastiin. Vertailumme tulokset osoittivat, että toteutuksemme skaalautuivat selvästi OpenNebulan tavallisia siirtomenetelmiä paremmin, kun käyttöön otettiin samanaikaisesti kymmeniä virtuaalikoneita. Toteutuksemme olivat kuitenkin normaaleja menetelmiä hitaampia, kun käyttöön otettiin vain yksi tai muutamia virtuaalikoneita. Tämä johtuu siirtoprosessin hallintaan kuluvasta ajasta.

Vertailimme myös eri virtuaalilevyformaatteja, koska levyformaatin valinta vaikuttaa virtuaalikoneen käyttöönottoon kuluvaan aikaan. Vertailimme tavallisia ja QCOW2-levykuvia sekä loogisia taltioita. Loogiset taltiot olivat nopeimpia peräkkäistä tietoa luettaessa ja kirjoitettaessa. Tavalliset levykuvat saavuttivat tässä tapauksessa 88% loogisten taltioiden kirjoitusnopeudesta ja 95% lukunopeudesta. Vastaavat lukemat QCOW2-formaatille olivat 86% ja 74%. Satunnaislukunopeus oli tavallisilla ja QCOW2-levykuvilla lähes sama, mutta loogisten taltioiden suorituskyky oli tässä suorittamassamme testissä selvästi näitä huonompi.

Jos pilviympäristöä käytetään niin, että useiden virtuaalikoneiden samanaikainen käyttöönotto on yleistä, toteuttamamme siirtomenetelmät nopeuttavat huomattavasti virtuaalikoneiden käyttöönottoa ja tehostavat sen resurssien käyttöä. Virtuaalikoneen levyformaatti tulisi valita niin, että saavutetaan riittävä suorituskyky ja pienin mahdollinen käyttöönottoaika.

Asiasanat:	pilviympäristö, pilvilaskenta, BitTorrent, multicast, OpenNebula, virtuaalilevy, virtuaalilevyformaatit
Kieli:	Englanti

Acknowledgements

I thank the supervisor for this thesis Jukka Nurminen and the instructor Pekka Lehtovuori for their comments on the various draft versions and pointing out interesting test cases. I would also like to thank the whole CE and later CS group for the ideas I got from the many coffee table discussions. I would especially like to thank Jarno Laitinen and Danny Sternkopf who took the time to read the different draft versions and give comments on those. Finally, being able to use a fairly large test system has been invaluable for good results, and I would like to thank everyone who made that possible.

Espoo, January 16, 2012

Risto Laurikainen

Abbreviations and Acronyms

ACK	acknowledgement
AS	autonomous system
CERN	European Organization for Nuclear Research
CPU	Central Processing Unit
GbE	Gigabit Ethernet
HTTP	Hypertext Transfer Protocol
I/O	input/output
iSCSI	Internet Small Computer Systems Interface
ISP	Internet Service Provider
LAN	Local Area Network
LCG	Large Hadron Collider Computing Grid
LVM	Logical Volume Manager
NAK	negative acknowledgement
NFS	Network File System
P2P	peer-to-peer
POSIX	Portable Operating System Interface for UNIX
RAID	Redundant Array of Independent Disks
RAM	Random Access Memory
RPM	Revolutions Per Minute
SAS	Serial Attached SCSI
SATA	Serial Advanced Technology Attachment
SCP	Secure Copy
SCSI	Small Computer System Interface
SSD	Solid-state Drive
SSH	Secure Shell
TCP	Transmission Control Protocol
UDP	User Datagram Protocol
VM	Virtual Machine
XML-RPC	Extensible Markup Language - Remote Procedure Call

Contents

Chapter 1

Introduction

Cloud computing provides a means of flexibly and easily allocating computing resources from one centralized pool to many different users[1]. This central pool is called a cloud and it is composed of many different interoperating technologies. At its core are a set of computers called hosts or nodes that provide the computing resources of the pool. Depending on the size of the cloud there can be anything from a few to tens of thousands of these computers. These computers are connected to each other with a fast local area network and typically also to the Internet with a high capacity backbone link, although it is possible to build a cloud that is entirely internal to a single organization.

On top of this infrastructure of computers and networks runs a layer of software dedicated to allocating resources from it for the users of the cloud. Virtualization is a key component of this software layer. This means that on the physical layer of computers and networks runs another layer of virtual machines and virtual networks which are implemented entirely in software. These virtualized resources are managed by a middleware component that launches and tears down virtual machines on the physical machines and connects them via the virtual networks.

1.1 Virtual machine

Software can be used to implement a virtual machine that runs inside a physical machine. The virtual machine can simulate real hardware or it can be a machine that has no physical counterpart. The Java Virtual Machine is an example of the latter. Just like a physical machine, the purpose of a virtual machine is to run code compiled for it[2]. If an operating system runs inside the virtual machine, this operating system is called a *guest operating*

system and the operating system of the physical machine hosting the virtual machine is called a *host operating system*.

1.2 Virtual machine images

The initial state of a virtual machine is stored in a file called an image. It is a virtual representation of the hard disk of a computer. To boot a virtual machine in a cloud environment, the image needs to be made accessible on a physical host that will act as the host machine for the virtual machine. The image is usually made available by copying it to the physical host. The cloud middleware stores a set of these images in an *image repository*.

Storing the state of the virtual machine in a single file has many benefits. The file can be used to easily encapsulate software and data that is required in order to complete the tasks of the virtual machine. Different users can have virtual machine images tailored to their specific field. These tailored images can be distributed through a central location that can be made available to the public. Storing the state in an image also means that it is simple to move between environments: the same image does not need to be restricted to a single cloud service. The image does not even need to be tied to a single middleware, as long as its format is supported by the *virtualization layer* operated by the middleware. Converting images between different formats is also often simple, which means images can be used with little effort on different virtualization layers as well.

1.3 Virtual machine deployment problem

One of the key benefits of the cloud computing paradigm is flexibility. This means that resources can be provisioned quickly and efficiently. If the process of deploying a virtual machine on a remote physical host takes a long time, some of this flexibility has been lost. The problem is not just a matter of having to wait for the virtual machine to become operational, since the deployment process itself takes up resources. Network resources are used to transfer data that is necessary to boot the virtual machine and disk space is taken up on the physical hosts to store that data. While the deployment is in progress, this resource usage does not produce any direct benefit. In other words, the resources are not used to work on the task for which the virtual machine is being deployed[3].

One of the fields that benefits from clouds is scientific computing. Scientists typically require powerful computers to run their applications in a

reasonable time. However, the need for computational power is not always constant. For example, a team of scientists might need to run simulations for two weeks and then not need any computational resources for a few months. It would be beneficial if the necessary resources the scientists need could be allocated as quickly and as efficiently as possible and then reallocated to other users once the scientists are done with them.

Minimizing the time and effort required to provision resources is particularly important for scientific computing. The tasks being performed in this field are computationally very demanding and will thus benefit from a large number of virtual machines working on a given task simultaneously. This is especially true for highly parallelizable tasks. However, deploying tens or even hundreds of machines to work on a problem can take a very long time. One solution is to simply deploy the machines once and then use them to work on a set of different problems, but this means losing the flexibility of cloud computing. Some of the tasks also use very large datasets, and if these datasets are encapsulated in the virtual machine image, then the deployment can be significantly slowed.

Typically the most time consuming part of deploying a large number of virtual machines in parallel is the transfer of the image files to multiple physical hosts. Typical image sizes are between hundreds of megabytes and some tens of gigabytes. When this is multiplied by the number of hosts which can be tens or even hundreds, the amount of data to be transferred becomes significant. Many different methods for achieving this task have been discussed in previous literature[3–9]. We classify these methods into four categories: centralized concurrent unicast, multicast, peer-assisted and on-demand distribution.

Centralized concurrent unicast distribution The naive approach is to start as many file transfers on the server serving the images as there are virtual machines to be deployed. However, this leads to transferring the same data multiple times, as the number of unique images to be transferred is typically much smaller than the number of virtual machines being deployed. It is in fact typical that most of the virtual machines in a single large deployment use the same image. For example, the worker nodes used by a distributed application for large scale computation usually all use the same base image. Computationally intensive distributed applications are also the type of application that benefits the most from large deployments.

Multicast distribution One better approach is to multicast each unique image so that it is transferred only once from the server and gets routed to all

the physical hosts involved in the deployment[4, 5]. This way each network link involved in the deployment needs to carry the data of the image only once instead of the sending network link having to carry it as many times as there are receivers. In practice, this method of transferring images also involves some retransmission of data to correct transmission errors and some control messages that are necessary for coordinating the transfer. However, the amount of this data is usually negligible compared to the size of the image.

Peer-assisted distribution Another approach is to divide the task of serving the image among the physical hosts involved in the deployment[5, 6]. There are two basics ways of doing this. One is for hosts that already have the complete image to start sending it to hosts that do not yet have it. The other is to start sending those parts of the image that have been downloaded to other hosts while the image download is still in progress. While in this approach the total amount of data transferred is the same as in the naive approach, the aggregate upload capacity available for the transfer is much higher as some or all the hosts involved also contribute to it.

On-demand distribution The fourth category is fundamentally different from the other three. Instead of moving the entire image to the physical host before booting the machine, its data can be accessed through the network as it is needed and any parts already accessed can be cached locally[3, 7–9]. The virtual machines can start booting right away instead of waiting, but disk I/O performance for parts that need to be read through the network will be lower. Even though this scheme still involves transferring the same data multiple times from one central location, the rationale behind it is that when only data that is actually needed is transferred, the total amount that needs to be transferred through the network is much lower. In other words, much of the data on the image files is actually never accessed.

Another aspect of the deployment process is the way in which the transferred data is made available to the virtual machine on the physical host. It can be made available directly as an image file, in which case any access to the data will involve two separate file systems – the file system that the image file is located in and the file system inside the image itself. Both of these filesystems will incur their own overhead on accessing the data. A more efficient method from a performance point of view is to make the data available as a specific reserved area on a physical disk called a *logical volume*, in which case only one filesystem is needed.

A third option is to create a new file or a logical volume called a *snapshot* that references another file or logical volume where the required data can be found. When changes need to be made to some part of the referenced data, that part is first copied to the snapshot and then changed. The original source of the snapshot remains unchanged. This pattern is called *copy-on-write*[10]. This third option is useful when, for example, multiple virtual machines using the same image need to be deployed on a single physical host. Using the copy-on-write pattern, it is possible to move the image to the physical host only once and then create a snapshot of that image for each of the virtual machines. The benefit comes from the fact that creating a snapshot is much faster than creating a true copy. However, the access performance of the snapshot is often reduced compared to a true copy[11].

As the choice of how the data is made available affects both the deployment time and the performance of the machine, it needs to be made based on what the virtual machine is going to be used for. If the virtual machine will need to process a large amount of data, then it is beneficial to make the data available so that access performance is maximized, even if this means that it takes more time to deploy the machine. If the virtual machine only needs to access a small amount of data but needs to do a lot of computation, then it is better to use the method that allows for the shortest deployment time, as disk performance is much less critical.

1.4 Research goals

The goal of this thesis is to evaluate the different approaches of transferring image data and making that data available to the virtual machines. This evaluation is done both in general terms and from the point of view of a specific cloud environment. This specific environment is a relatively small cloud system with 72 physical machines. The aim is to find the best method of deployment given the size of the system and the performance of the network that is used for the deployments.

The basic transfer methods we use are BitTorrent[12] and multicast[13]. The specific program we use for the BitTorrent transfers is rTorrent[14] which is based on an open source BitTorrent library called libTorrent[15]. To test multicast transfers, we evaluate two different programs – UDPcast[16] and UFTP[17]. We also run disk I/O benchmarks on virtual machines that have been deployed with different means to evaluate the differences between disk access methods.

In order to be able to use the file transfer programs mentioned for virtual machine deployment, we implement a set of *transfer managers* for a middle-

ware software called OpenNebula[18]. The transfer manager is a component of the middleware software that is responsible for managing the transfer of the images to hosts from the image repository and doing the necessary preparations on the hosts so that virtual machines can boot using the image data. In OpenNebula, the transfer manager is implemented as a set of scripts that call external programs that do the actual work of transferring images and deploying them on the host.

We make a distinction between the two tasks of the transfer manager. We call the way in which the transfer is handled the *transfer method* and the type of image data that is used and the way it is processed on the hosts the *disk access method*. The choice of the transfer method affects the time it takes to deploy a virtual machine and the choice of the disk access method affects both the deployment time and performance of the virtual machine. We attempt to answer the following research questions:

- What is the best transfer method in this environment for deploying multiple virtual machines simultaneously?
- In which situations should each of the different disk access methods be used?

1.5 Structure of the thesis

The rest of this thesis is organized as follows. Chapter 2 gives an overview of the related research and explains the key concepts used in this thesis. Chapter 3 describes the environment used for evaluating the implemented transfer managers. The way the different elements of the environment are connected and the hardware of the physical servers in the environment are described. Chapter 4 gives a detailed description of the functionality of the implemented transfer managers. The algorithms with which they achieve their goals are described. Chapter 5 describes the procedures that were used to benchmark the transfer managers. The results of the benchmarks are in Chapter 6. Chapter 7 summarizes and discusses the findings of the evaluation and contains usage recommendations about the different transfer and disk access methods. The source codes for the transfer managers that we implemented can be found in Appendix A.

Chapter 2

Background

In this Chapter, we look at previous research into virtual machine deployments and file transfers in general. We also introduce some of the concepts that are central to this thesis.

2.1 OpenNebula

OpenNebula[18] is a set of tools for managing a cloud system. A set of physical resources like networks and hosts are defined in its configuration and it then uses these for deploying virtual machines. The design is modular, and various components can be replaced with new ones. One of the components that can be replaced is the transfer manager. This component defines the way in which data necessary for deploying a virtual machine is transferred to a host from an image repository that can reside on the machine running OpenNebula or a separate server machine. At the time of this writing, the default distribution of OpenNebula contains a small set of transfer managers that the administrator of the system can choose from when installing a new cloud environment. These transfer managers are fast and fairly efficient for deploying a few hosts at a time, but larger deployments will inevitably take much longer.

SCP transfer manager

The SCP transfer manager uses the Secure Copy (SCP)[19] program to move virtual machine image files. For each new deployment, a new SCP process is started. This method of transferring images is very inefficient. For example, if the same virtual machine image needs to be transferred to 100 hosts, then the image repository host must send the same data 100 times.

Shared transfer manager

When using the shared transfer manager, the directory containing the virtual machine images will be exported to the virtual machine hosts using NFS. When a machine is deployed, the image is either transferred through NFS to the host or a symbolic link is created if the image in question is not cloneable. As with the SCP transfer manager, the image repository becomes a bottleneck.

LVM transfer manager

The LVM transfer manager is similar to the SCP transfer manager, but instead of simply transferring the image file to the virtual machine host, it writes its content to a logical volume from which the machine then boots. As was discussed earlier, this reduces the I/O overhead of the virtual machine. However, the transfer of the image over the network is not improved in any way.

Virtual machine lifecycle

Each managed virtual machine has a lifecycle composed of a number of states, and OpenNebula keeps track of those states using a state machine[20]. When a new virtual machine is created, it starts in the "pending" state. At this point the virtual machine is not tied to any physical host. A scheduler component will attempt to find available physical resources for those VMs that are in the pending state.

Once the resources are found for a VM, it will be move to the "prolog" state. In this state, the persistent state of the virtual machine is transferred to the physical machine. It can either be transferred in whole or the virtual machine can boot using an image stored in a central location and only keep track of the changes made to the persistent state. The transfer manager handles the transfer that takes place in the "prolog" state.

Once the transfer manager finishes its task, the virtual machine moves to the "boot" state and from there to the "running" state. At this point a connection to the virtual machine can be established. Once the VM is no longer needed, it can be deleted or gracefully shut down.

There are also other states related to saving the state of the machine, migrating it to another physical machine, suspending it and states related to various errors. For the purposes of this thesis it suffices to only look at the states from "pending" through "prolog" to "boot" and "running".

OpenNebula frontend

This is the physical machine that runs the OpenNebula software. When we refer to "the frontend" in this thesis, this is referring to this machine.

2.2 Transfer methods

In this Section, we describe the specific transfer methods used in this thesis and in the literature. These are methods that transfer whole images and have been proposed as ways for more efficient use of network resources.

BitTorrent

BitTorrent[12] is a protocol for rapid distribution of files over local or wide area networks. It leverages the upload capacity of downloading hosts to speed up transfers and to scale to a large number of hosts. Files that are to be served are divided into multiple pieces. These pieces are then served to a number of receivers from the initial source of the file. Once a receiver has finished downloading a given piece, it can start uploading it to the other receivers. This increases the total upload capacity of the system.

BitTorrent uses separate tracker software to allow receiving hosts to find other receiving hosts. The tracker is a server that has information about which hosts are downloading which files[12]. More recently, distributed ways of tracking other downloading hosts have been added, making the tracker an optional component. One example is distributed hash tables[21–24].

Metadata about the files to be shared is contained in a so called *torrent file*. This file contains hashes for the pieces of the files. These can be used to check the integrity of pieces before distributing them to other hosts. It also contains information about the tracker.[12]

Previous research on the scalability of BitTorrent has shown that it scales well to a very large number of simultaneous peers[25–27]. It works well in very heterogeneous environments. For example, Izal *et al.* studied one publicly available torrent that was downloaded by a wide variety of users using many different connection types and found out that the users were mostly able to utilize the bandwidth available to them effectively[25].

Multicast

Multicast is a routing scheme in which data is sent only once by the sender and is then forwarded to a group of receivers[13]. Unlike in broadcasting,

packets that are multicast are only forwarded to hosts as necessary and not to all possible hosts.

Any transfer protocol that is used for file transfers must provide reliability and correct ordering of data. As multicast is typically a best effort protocol, these functions need to be implemented separately. There are various methods for providing reliable multicast, and the best way depends on the context in which multicast is being used.

The scalability of any multicast file transfer tool depends on the reliability mechanism it uses. One very scalable reliability mechanism is to send redundant data along with the files to be sent. Packet losses can then be recovered by using the redundant data. This scheme scales similarly to unreliable transmission, as it is almost exactly the same case except for the slightly increased amount of transmitted data. However, if more packets are lost than can be recovered using the redundant data, there is no way for the receivers to notify the sender that retransmission is necessary, and the data is irrecoverably lost.

For better reliability guarantees, clients should be able to notify the sender about packets that have been received with positive acknowledgements (ACKs) or about packets that were lost with negative acknowledgements (NAKs). Using ACKs quickly leads to scalability problems, as the sender needs to process all the incoming ACKs from the receiving nodes[28]. Since the number of incorrectly received packets is typically much lower than the number of correctly received packets, using NAKs lessens the burden on the sender and is thus a more scalable solution. It can be further optimized by using NAK avoidance: when a receiver notices a packet has been lost, it waits for a random time before multicasting a NAK to all the receivers and the sender. If it receives a NAK for the same packet from another receiver while waiting, it does not send the NAK itself[28].

Binary tree distribution

In binary tree transfers, when a host has finished downloading a file, it can start sending it to other hosts, thus leveraging their upload capacity. In this regard the idea is similar to what is used in BitTorrent. However, the hosts are unable to start sending out the files before they have been fully downloaded and the last nodes to receive the file (the leaves of the tree) do not contribute to the upload capacity[12]. One implementation of this algorithm is a Python script called scp-wave[29].

2.3 Virtual machine disk optimizations

There are various methods that can be used to speed up the deployment of virtual machines that are not directly related to the transfer methods. These include compressing the virtual machine image to reduce transfer time, creating the images as so called *sparse files* that do not explicitly store empty blocks and methods to greatly reduce the time required to make copies of files.

Copy-on-write

Copy-on-write is an optimization scheme that avoids unnecessary copying[10]. When data is to be copied, a reference to it is created instead. When a part of the data is written to, a true copy of that part of the data is made so that the original referenced data remains intact. By not copying all the data at once, this scheme makes it possible to nearly instantly create light weight copies of large files like virtual machine images. This is a useful feature for those transfer managers presented in this thesis that rely on an intermediate copy operation. The intermediate copy is discussed in Chapter 4.

QCOW2 disk image format

QCOW2 is a disk image format for the QEMU processor emulator[30]. As OpenNebula uses QEMU, QCOW2 is one of the disk image formats available for it. It has many features that are unavailable in raw disk images. It supports compression, encryption, copy-on-write copies and multiple snapshots inside one image file. However, the I/O performance of a virtual machine deployed using a QCOW2 image is less than what is available when using a raw image[31].

For the purposes of this thesis the most important benefits of QCOW2 are the smaller image files and copy-on-write copies which enable faster deployments. The benefit of the shorter deployment times must be weighted against the lower performance. The task for which a particular virtual machine is to be used must be taken into account as well, since many tasks are not I/O intensive and are thus only slightly affected by the slower performance of QCOW2.

2.4 Previous research

Research into efficient file distribution in general has a longer history than the more specific research area of virtual machine deployments. Content distribution networks[32–34] have been used to distribute data so that it is closer to the users that access it, making the accesses faster and less taxing for the network. Various peer-to-peer (P2P) schemes[12] have been proposed and implemented that aim to distribute the task of transferring a file to a set of hosts instead of one central host. Reliable multicast[28, 35–40] has been researched as a way to transfer files without overloading the network. Image deployment research builds on the foundation of the more general literature and adapts the techniques to its specific use case. The techniques used in the literature can be roughly divided into the categories introduced in Chapter 1: multicast, peer-assisted and on-demand distribution.

2.4.1 Multicast distribution

Frisbee is a system for distributing disk images that uses reliable multicast and file system aware compression[4]. Frisbee is not designed to deploy virtual machine images but to deploy the operating systems of physical machines. However, many of the same principles still apply. The tests conducted for the paper show the system to be fast and efficient at simultaneously deploying tens of machines. A 3067 MB file system can be deployed to 80 machines in a little over 100 seconds using file system aware compression.

The authors note that the application domain has many features that allow for aggressive optimization: the entire system is under the control of the administrators, all of the resources on all of the machines being installed are dedicated to the imaging task while the installation is in progress and there is a fast network connection to all the machines[4].

In virtual machine image transfers, the administrators also have full control of the environment and there is a fast network connection between the machines, but not all the resources on the physical hosts can be assumed to be available for the deployment process. These machines may already be running other virtual machines that may be consuming most of the available resources. Having these resources fully available is useful, as it allows the deployment process to use most of the RAM for caching purposes and the CPU for compression and possibly encryption if this is required for security reasons.

2.4.2 Peer-assisted distribution

The two basic approaches to peer-assisted image distribution discussed in Chapter 1 – sending complete images between peers and sending pieces of images between peers – have been studied in the image transfer literature. More specifically, a tree based algorithm and BitTorrent have been compared in two separate papers[5, 6].

In the tree based algorithm, the complete image is first sent to a few nodes, which will send it to a few other nodes and so on, until all the target nodes have the image[5, 6]. This way the distribution is based on a tree structure. The transfer accelerates as it gets closer to the leaf nodes of this tree as more and more peers start sending, providing a logarithmic speedup. The specific implementation tested in the papers is scp-wave[29].

Instead of using a tree distribution pattern, distribution in BitTorrent is based on each peer communicating to a random set of peers[12]. This is potentially more efficient, as the leaves in a tree structure do not contribute to the upload capacity[12]. As a large part of the nodes in a tree are leaf nodes, a significant amount of upload capacity is lost. As files are also split into pieces when using BitTorrent, peers can start contributing to the upload capacity sooner than if they had to wait to receive the complete file.

Another program similar to BitTorrent that splits files into pieces for transfer is scp-tsunami[41]. It is a successor to scp-wave, but is functionally closer to BitTorrent. However, neither program appears to be actively maintained, as the latest versions are over a year old at the time of this writing[29, 41].

Wartel *et al.* [6] tested the performance of BitTorrent and scp-wave for large scale deployments of virtual machines in a cloud environment called *LxCloud*[42], which is a part of the Large Hadron Collider Computing Grid (LCG). The CERN environment described in the paper is similar to the environment used as a basis for evaluation in this thesis and described in Chapter 3. OpenNebula is used in both as the middleware software. The characteristics of the hardware are also similar: both have multiple identical servers connected by high bandwidth, low latency links[43]. Wartel *et al.* note that BitTorrent requires some tuning to work well in this type of environment. This is to be expected, as BitTorrent is designed to run in a far more diverse environment with highly variable bandwidths, latencies and client hardware capabilities.

BitTorrent was found to be significantly faster: deploying 462 virtual machines from a 10 GB image took 23 minutes, while for scp-wave it took 50 minutes. However, scp-wave was still seen as a potentially viable alternative due to its simplicity of use[6].

Schmidt *et al.* [5] compared the deployment times of unicast, binary tree, BitTorrent and multicast image transfer methods. They concluded that while multicast is fastest overall, it is not suitable for deployments outside the local network. For this task they propose the use of BitTorrent, which was the second fastest option in their tests. However, their reported test results are vague as they report only the total number of cores on the physical machines that they deploy to and not the actual number of the physical hosts. There were 80 cores which would suggest that the number of physical hosts was most likely between 10 and 40 assuming a single machine has 2 to 8 cores. The number of virtual machines deployed was the same in all the tests. In their tests the average time to deploy an 8192 MB image to the test machines using BitTorrent was 465 seconds and 383.5 seconds using multicast, while the average time was 1634.1 seconds using unicast.

2.4.3 On-demand distribution

Nicolae *et al.* [7] propose a fundamentally different way of rapidly deploying multiple virtual machines. Instead of waiting for the entire image to be transferred before booting the virtual machine, they propose a system where content is transferred as needed from the image repository while the machine is running. This way the machine can boot right away.

The system works by intercepting I/O calls from the running machine and forwarding them through the network. Any data that has been previously read through the network is cached locally so that any subsequent reads to that data will not go through the network but will instead be retrieved from the faster local cache. This disk access method does introduce a performance penalty due to the necessity of initially reading data through the network and from one central host. The authors of the article argue that this performance penalty is outweighed by the rapid deployment time and that the total execution time of computational tasks is still reduced. While their test results support this, the image deployment tool they compare their implementation against – called TakTuk[44] – may not be as fast as the fastest deployment tools for transferring the entire image – namely those based on BitTorrent or multicast. TakTuk is a tool for efficiently scheduling remote executions. As far as we can see, it does not make the transfer itself any more efficient.

SnowFlock is another system that similarly only sends data that is needed through the network[9]. It uses an abstraction called VM fork that is similar to the fork operation for Unix processes. This means that the state of the parent VM gets cloned to the child VM. The system also uses multicast to transmit replies to page requests from the cloned VMs. The rationale behind this is that different VMs often request the same data. Multicast can

therefore avoid unnecessary transfers.

Wu *et al.* [3] propose a solution to the virtual machine performance issue inherent to on-demand distribution. Instead of having a single location where virtual machines can access the image, the image is distributed to multiple hosts. Only a part of the virtual machines access any one of these copies, which reduces contention. They use the term *cloud time* to describe the time required to deploy a set of virtual machines and have them cooperate to complete a task. In other words, they attempt to minimize the time to deploy a set of virtual machines while still keeping task runtime as low as possible.

They test three different methods for distributing the images. The first is to pre-distribute the images to a number of hosts before any of the virtual machines start. They also propose two optimizations. One is to store the image in memory instead of the disk to make accessing it faster. However, one issue not addressed by the paper is that this is not a viable option in many environments where images sizes can be far larger than the size of memory. The other optimization is to store a given copy of the image on a host that is in the same subnet as the virtual machines using that copy. By storing the image closer to the virtual machines accessing it, fewer hops are necessary to access the information.[3]

The second method acknowledges the fact that it is difficult to predict how often images are going to be accessed and where the accesses will come from. All virtual machines initially boot using the same central image. The load on that image and the read performance of the virtual machines is monitored, and new image copies are made if performance is not adequate. Some virtual machines must then be redirected to use the copies of the image instead of the original. The goal is to avoid the initial overhead caused by pre-distributing the image while being able to respond to a higher access load if necessary.[3]

The third method attempts to exploit the fact that much of the image will remain unread throughout the lifetime of a virtual machine. Empty *pseudo images* are created in the beginning instead of full image copies. As with pre-distribution, each of these images serves a subset of all the cloned virtual machines. When these machines access data on the pseudo image, that part of the data is copied over from the master image and cached. Data that is never read does not need to be transferred through the network.[3]

The findings of the paper indicate that the distribution of shared images does improve performance and decrease overall application completion time. Pre-distributing the images to a number of nodes or using pseudo images are found to give the most gain compared to one central master image.[3]

The authors' solution of having multiple shared images can be seen as

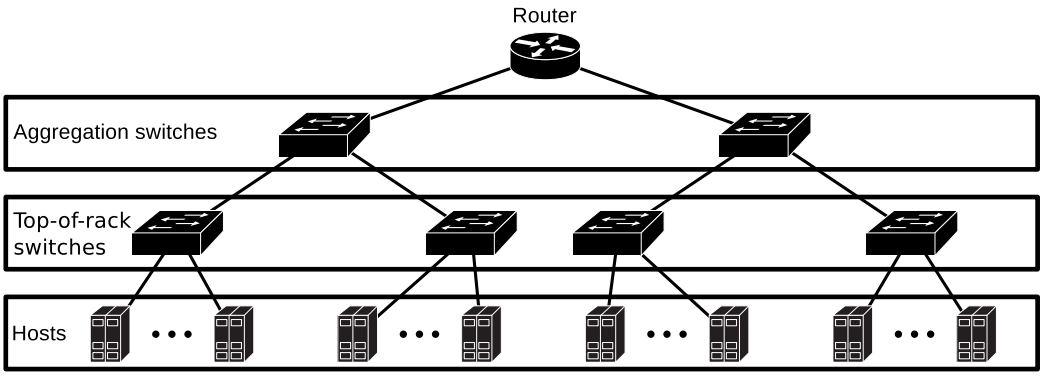

Figure 2.1: A simplified view of the network topology of a datacenter network[45].

a hybrid of the on-demand distribution method and regular image transfers to local disks. Regular image transfers can be seen as a special case of the image distribution where the number of copies is simply the same as the number of virtual machines being deployed. By making more than one copy but still less copies than there are running virtual machines, this approach would seem to be a viable compromise between virtual machine performance and deployment time.

2.4.4 Network topology effects

While datacenter networks have low latencies and high bandwidth, not all of the links are equally fast. The more switches there are between two machines, the higher the latency is between them. As modern datacenters can have thousands of machines and switches have a limited number of available ports, many switches will be needed. These are organized in a hierarchical topology illustrated in Figure 2.1. Some connections between two machines will traverse only one switch while others will traverse several. The higher a link is in the hierarchy the more bandwidth it must have to provide optimal performance[46]. The test environment used in this thesis and shown in Figure 3.1 follows this same hierarchy, but in a smaller scale.

In order to maximize the performance of distributed software, the topology of the network must be taken into account. As much of the traffic as possible should be between closeby hosts, and connections traversing multiple switches should be avoided if possible. One example of this relevant to image transfers is the peer selection in BitTorrent which is discussed in Chapter 6.

This issue has been studied in the context of public BitTorrent swarms where peers are connected through multiple geographically distributed internet service providers. The basic principle is the same: peers are inside highly interconnected islands, and the connection between the islands is a bottleneck. In other words, the connections between customers of the same ISP are typically faster and consume less resources than connections between customers of different ISPs. Research into reducing traffic generated by P2P networks between ISPs has concentrated on reducing operational costs, but at the same time the average download speeds of peers have also increased[47–50].

Bindal *et al.* [47] suggest multiple different methods for determining peer closeness. Internet topology maps, autonomous system (AS) mappings and IP range information published by ISPs can be used by trackers to determine which peers are close to each other and to advertise peers to each other accordingly. The ISP can also append a new HTTP header containing locality information to BitTorrent packets. These solutions require support from either the tracker or the client software.

Another suggested solution that does not involve changing either is for the ISP to use dedicated devices to modify peer advertisements from trackers so that peers within the ISP are selected instead of outside peers. In order to configure the device, the ISP can use its knowledge about which peers are connected to its network to make the closeness determination.[47]

Aggarwal *et al.* [48] describe an *oracle* run by the ISP that can be queried by P2P clients to determine a ranking between members of a set of potential peers. Those peers that are close to the querier will be ranked higher than those that are not. The ISP can use its knowledge about its own network to determine the rankings. Various metrics are suggested in the paper, such as whether a peer is inside the same AS as the querier and if the querier is not within the same AS, the number of AS hops required to reach the peer from the querier.

Choffnes and Bustamante [49] use information from content distribution networks to calculate distances between peers. Content distribution networks must be able to determine the replica servers closest to a given peer. If two peers are redirected to the same replica server by the content distribution network, those peers are most likely close to each other. Multiple redirections per peer are observed, and the more similar the redirection behavior of two peers is, the smaller the distance should be between them. The redirection behavior of a peer is represented as a vector, and the closeness of two peers is defined as the *cosine similarity* of those vectors.

Chapter 3

Test environment

The system on which the transfer manager tests were run is an extension to an existing cloud system operated by CSC – IT Center for Science Ltd. As this was a recent procurement that had not yet been taken into use, it provided a good environment for testing. The hosts were dedicated for test purposes and there was no other load on the system to skew the results. However, as the system was still being built, some features of the final system were missing. Most notably, the virtual machine hosts only had 1 gigabit Ethernet (GbE) connections (see Section 3.2) to the switches instead of 10 GbE connections.

3.1 Test hardware

The frontend machine has two 6-core Intel Xeon X5650 2.66GHz processors, 72 GB of memory and six 600 GB 15000 RPM SAS hard disks. All of the virtual machine hosts also have two 6-core Intel Xeon X5650 2.66GHz processors, but the amount of available memory varies between the hosts. 44 hosts have 24 GB while 24 hosts have 48 GB. The remaining four hosts have 96 GB of memory, though these were not used for the tests as they also have faster hard drives than any of the other nodes, and this may have skewed any results whose outcome depends on disk I/O performance. The regular hosts have two 7200 RPM SATA hard disks. The virtual machine images and logical volumes are stored in a striped RAID 0 array on these disks to improve performance.

3.2 Network layout

The network layout of the test system presented in Figure 3.1 is simple. All the hosts are connected to the same network. There are two 48 port switches.

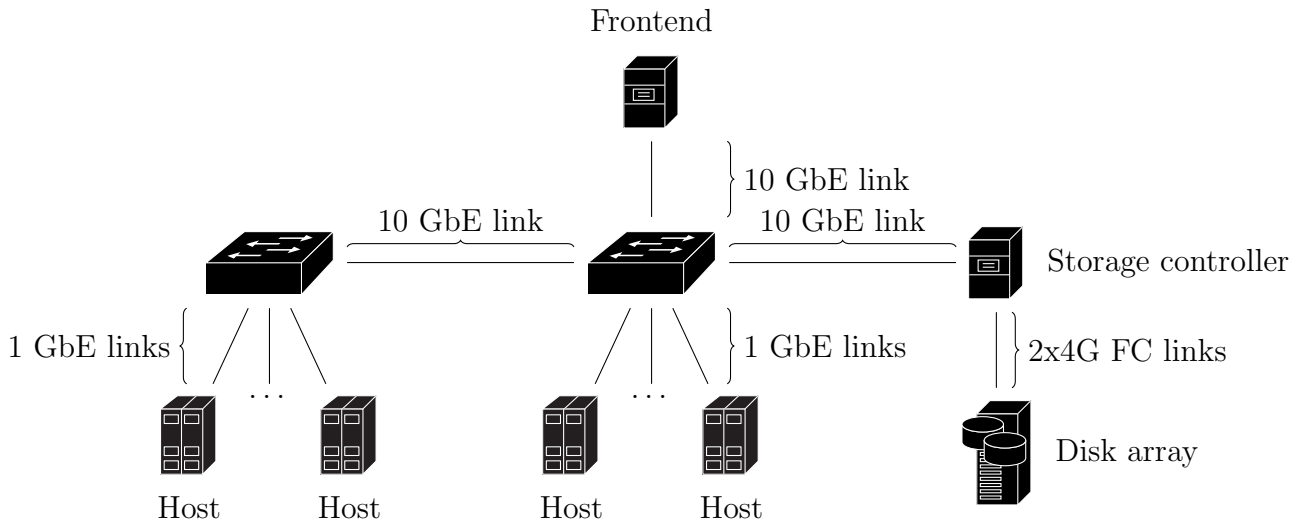

Figure 3.1: The network layout of the test system.

The frontend is connected to one of these switches with a 10 GbE link. There are 72 virtual machine hosts. Half of the these are connected to the same switch as the frontend and the other half is connected to the other switch. The connections between the hosts and the switches are 1 GbE connections. The switches are connected to each other using a 10 GbE connection.

The system is also connected to a disk array through a separate storage controller as is shown in Figure 3.1. The storage controller is a NetApp V3240, and it is connected to the disk array using two 4 gigabit Fibre Channel (FC) links. The storage controller contains 512 GB of SSD cache. It is possible to use this disk array as an image repository. However, it was only connected to the system at a late stage of testing, so to maintain comparability between the tests, most of the transfer manager tests used the local disk of the frontend as the image repository. Only the NFS transfer manager was tested when using the disk array as an image repository.

The simplicity and the degree of control over the system makes it possible to use many different transfer mechanisms. Multicast in particular is easier to set up than it would be for a more complex system with, for example, connections to remote hosts through the Internet. There is no need for multicast routing, as all the hosts can be reached in a single hop. It would even be possible to use broadcast to transfer virtual machine images.

Chapter 4

Implemented transfer managers

In this Chapter, we describe the transfer managers we have implemented for OpenNebula. We explain how simultaneous image transfers to multiple physical hosts are coordinated between different instances of the same transfer script, processes related to the transfers and how these processes run in the image repository and on the physical hosts used to host virtual machines.

An OpenNebula transfer manager comprises several scripts that are used in different stages of a virtual machine's lifecycle. For example, there are scripts for moving an image from one location to another, deleting a virtual machine, creating empty virtual disks and swap space. To implement the transfer managers described here, we have written implementations of the *clone* script. This script is used to copy images from a central location to one or more physical hosts. As the implementations of the other scripts do not affect the deployment speed problem, they have been reused from the SSH transfer manager that is a part of the basic OpenNebula package. The source codes for our implementations can be found in Appendix A.

4.1 BitTorrent

There are a number of features that a BitTorrent client must have for it to be usable in the context of an OpenNebula transfer manager. Because administration is largely done using a command line interface, it should be possible to interact with the selected BitTorrent client using one. For automation, it must also have some type of interface that accepts commands from scripts without the need for manually interacting with a user interface.

Using these criteria, we selected rTorrent[14] as the client used for the BitTorrent transfer manager. It has a command line interface which can be used for debugging and solving potential problems. Commands can also

be issued from scripts through an XML-RPC interface. The client supports running scripts based on events. For example, a script can be run when a download finishes to trigger further actions. RTorrent can also monitor a directory for new torrent files and start downloading the files associated with any new torrent files that appear in that directory.

Another necessary component for the BitTorrent transfer manager to work is the tracker. For this task, we chose opentracker[51] due to simple configuration, its lightweight nature and support for a clustered configuration. Using a clustered configuration means running multiple instances of opentracker on multiple hosts so that all the instances synchronize the torrents being tracked among each other. This provides redundancy. However, the current implementation is based on using only a single tracker.

RTorrent runs constantly on both the image repository machine and the virtual machine hosts. The image repository host acts as an initial seed for all the images in the image repository. The necessary torrent files are generated and stored on the image repository host.

When the transfer manager clones a virtual machine, it first checks to see if the correct image is already on the host where the VM is being cloned. If it is, it creates a local copy of it instead of transferring the image through the network. This is the case when the image has already been transferred to the host by a previous deployment, but it has not yet been deleted.

If the image could not be found on the host, the transfer manager triggers a download by sending the corresponding torrent file to a directory being watched by the rTorrent client running on the host. The client notices the new torrent file and starts the download. Once the download finishes, it runs a script that creates a file notifying the transfer manager script about a finished download. The existence of this file on the remote host is being polled at regular intervals by the transfer manager script. When it sees that it has been created, it knows that it can find a complete image file and copies this image file for deployment under a subdirectory of the VM directory.

The transfer manager handles deleting virtual machines by only deleting the copy of the image that the machine booted from. The original transferred image is left on the host so that further clone operations of that image to that host only need to do a local copy operation. This also means that hosts that have a given image can continue seeding it to other hosts, distributing the load of image transfers. It is possible that at some point these old images would take up so much space that deploying any new images becomes impossible. This case is handled by a separate Nagios[52] event handler (see Section 4.4).

4.2 UDPcast

UDPcast[16] is a multicast file transfer tool. It runs a server process called *udp-sender* on the machine sending files and a receiver process called *udp-receiver* on each of the machines receiving files. Multicast groups are automatically negotiated between the sender and the receivers using broadcast messages. UDPcast can write to a pipe instead of a file. This is useful for e.g. writing directly to LVM partitions or writing to multiple files at once. It can automatically regulate its transfer speed and the regulation mechanism can be replaced with a customized version.

To ensure reliable transfers, the server in UDPcast first sends a set of packets called a slice and then asks for an acknowledgement from the clients for those packets. The clients will reply with either a list of packets to retransmit or a message notifying the server that no packets were lost. Once the server has received an acknowledgement from all the clients or a timeout has been reached, it will begin transmitting the next slice.[53]

The UDPcast transfer manager uses shared files and locks to coordinate multiple simultaneous deployments. The receiver processes need to know where to write the file being received. This information is communicated to the receiver processes using shared files: each instance of the cloning script writes the full path of the disk image to be written on the virtual machine host into a shared file unique to that host.

The frontend should only have a single server process running and the virtual machine hosts should only have a single receiver process running. Locks are used to ensure this. Locks are also used to ensure mutual exclusion when accessing shared files. There are four different locks: an image specific transfer lock, a global UDPcast lock, a host specific receiver lock and an image specific destination list file lock.

Whichever instance gets the image specific transfer lock starts the udp-sender process while the other instances wait. The transferring instance also needs to acquire the global UDPcast lock before starting a transfer. This is to ensure that only a single udp-sender process is active on the frontend at any one time.

The host specific receiver lock is used to ensure that only one instance of udp-receiver runs on each host that an image is being deployed to. The image specific destination list file lock is used to ensure that only one instance at a time writes to a shared file of deployment locations. This lock is also acquired by the transferring instance when it starts a transfer to ensure that instances of the cloning script that start while a transfer is in progress do not write a destination to the old shared file that has already been used to determine

the locations to write to and will no longer be read by the receivers. Any instances starting late will instead have to wait to acquire this lock, so that they write instead to a new shared file that gets used in a new deployment that starts after the one in progress has finished.

Unlike the other new transfer managers, the UDPcast transfer manager avoids the additional copy operation after the transfer by writing simultaneously to multiple locations using the *tee*[54] command. The output of udp-receiver is sent to a pipe instead of a file. Tee reads from this pipe and writes to the locations listed in the shared file that each instance added a destination line to.

Unlike the UFTP and the rTorrent transfer managers, the UDPcast transfer manager does not require any processes to constantly run on the hosts or the image repository. Any necessary processes are launched from the cloning script. This makes it simpler from an administrative point of view and means there are fewer components that can break down.

4.3 UFTP

UFTP[17] is a multicast file transfer tool. A server process called *uftp* runs on the frontend and sends data to receiver processes called *uftpd* on the virtual machine hosts. The server sends data at a fixed rate. Reliability is achieved through *negative acknowledgements* (NAKs). The server initially sends a file sequentially from start to finish without any retransmissions. When a receiver notices that it is missing data that it should have received, it sends a NAK to the server. The server collects these NAKs from all the receivers and when the first round is finished, it begins retransmitting a union of all those packets it received a NAK for. This process is repeated until all the receiving hosts have all the packets.

Locks and shared files are used for coordination between different instances of the cloning script. The server process takes as input a shared file with a list of hostnames. Before the server is started, each instance of the cloning script writes the hostname to which the image is being deployed into this file. In other words, this shared file defines the multicast group.

As with UDPcast, there is a lock to ensure that only one of the instances tries to start the server process. There is also a global lock to make sure only one instance of uftp runs at any one time. This is necessary as the transmission rate is set to a reasonable fixed maximum, and more than one process transmitting simultaneously at this rate would exceed the capacity on either the sending or the receiving links. This would lead to packet loss and inefficient transfers. The shared file containing hostnames is locked while

the transfer is in progress.

The receiver process runs all the time on the virtual machine hosts. It writes any files received to the root of the virtual machine directory. From there, the image files are copied to their final locations to be deployed after the transfer has finished. If an image has previously been transferred to the host and the file still exists at the root of the virtual machine directory, it can be copied directly from the local copy, thus making the actual transfer unnecessary.

4.4 Image cleanup script

The feature of the BitTorrent and UFTP transfer managers to leave virtual machine images on the hosts means that there needs to be a mechanism to clean up these images to avoid a situation where the old images take up so much space that new machines cannot be deployed. We implemented a script for this purpose that is launched by a system monitoring tool called Nagios[52].

Nagios is used to monitor the free space on the physical hosts hosting the virtual machines. When it sees that the free space on a host drops below a certain threshold, it triggers the script that then cleans up unneeded images on that host. These are images that are not related to any transfer in progress and are not being used by any of the currently running virtual machines.

The script will also remove any other files related to the transfer of an image file that are not image files. For example, the script will delete torrent files and files that notify the frontend about a finished transfer. If BitTorrent is used, the script removes the transfer related to the image from the torrent client on the virtual machine host.

4.5 Parameter selection

Various parameters that affect transfer speed can be adjusted for each transfer method. For BitTorrent, these parameters include, for example, the piece size of a torrent and the maximum number of simultaneous connections on a client. For UFTP, the transfer rate, various timeouts and the send and the receive buffers can be adjusted. Before evaluating the transfer managers, we experimented with different parameters to find out a reasonable set that would provide good performance and reliability. For the multicast transfer managers this was mandatory, as the wrong parameters would have led to timeouts, dropped packets and poor performance. For BitTorrent the

changed parameters only provided slightly better performance, as the default settings were already quite good.

The default settings for UFTP are not usable in the context of a fast data center environment. As there is no automatic control for the transmission rate, it needs to be set manually. By default it is set to 1000 kbps, which is two orders of magnitude slower than the rates that can be expected to work reliably in the network. The transmission rate that provided the shortest transfer times was determined through experimentation to be 800000 kbps. However, tuning the transmission rate proved to be very difficult, as the optimal setting depends not only on the network but also on other factors, such as the operating system send and receive buffers.

Unlike UFTP, UDPcast has its own rate limiting logic. This means that there are far fewer parameters to adjust. The most important of these is the slice size. It is the number of packets sent to receivers before querying them to find out which packets were lost[53]. A slice size of 24 packets was chosen, as packet loss was greatly reduced by doing so.

A piece size of 4 MB was initially chosen for the torrent files used by BitTorrent. This was to avoid overhead from having to handle a lot of small pieces. With 4 MB pieces, the approximately 10 GB image used for testing splits into 2501 pieces. Further testing revealed that smaller piece sizes would have in fact been slightly better for the average deployment time of the virtual machine, but that the difference is very small. The results for these tests are in Section 6.4.

Chapter 5

Test methods

In this Chapter, we describe the tests used for evaluating the different aspects of the transfer managers. Deployment benchmarks were used to test the efficiency of transferring image data to multiple physical hosts simultaneously. Disk I/O benchmarks were used to compare the performance of different disk access methods. Finally, the effect of piece size and network topology on BitTorrent transfers were tested. The results for the tests described can be found in Chapter 6.

5.1 Deployment benchmarks

The total time used to deploy a number of virtual machines from the image repository to the hosts was measured by reading data from OpenNebula's virtual machine log files. Each virtual machine instance has its own log file, and OpenNebula writes a timestamp to the beginning of this file as the virtual machine's state changes from the pending state to the prolog state.

The cloning script of each transfer manager was modified so that the last command to run writes another timestamp to the log file. The total time to deploy was taken as the difference between the initial timestamp from the log of the first machine to go to the prolog state to timestamp written by the last command of the cloning script of the last virtual machine to go to the boot state.

The deployment time of a single virtual machine was measured using these same timestamps. It was taken as the difference between the first and the last timestamp written by the deployment. In other words, the time spent in the prolog state was measured for each virtual machine.

The virtual machines spend a short but unpredictable amount of time in the pending state before being deployed, but this time was not counted

in the measurements as it is not related to the transfer manager logic but rather the logic of the virtual machine scheduler. It should also be noted that only the time taken to transfer the images and not to actually boot the virtual machines was measured. This method of measurement was chosen so that the comparison would not be affected by the variability in deployment time introduced by the scheduler or the possible variability of boot times. The tests were repeated 10 times and the results averaged to mitigate errors produced by, for example, system processes interfering with the tests.

Certain modifications to the scheduler parameters of OpenNebula were necessary in order to get the best possible performance out of the transfer managers. The aim of these modifications was to deploy as many virtual machines as possible concurrently and to have as many machines as possible go to the prolog state at the same time. In OpenNebula 3, the maximum number of concurrent deployments is by default set to 15. This is a hardcoded limit in one of the source files. This number has most likely been chosen to limit the number of concurrent SCP processes. However, the problem of concurrent processes does not exist with the other transfer managers. The performance of the multicast transfer managers in particular is severely reduced by this low setting. The limit was thus raised to 60. To allow for as many virtual machines to go to the prolog state simultaneously, the maximum number of VMs dispatched was raised to 70.

The traffic measurements were done using readings from *ifconfig*. One reading was taken on the frontend and the hosts before a deployment test and another one right after the test. The difference between these two was taken as the amount of data transferred on that host between the readings. One possible shortcoming of this method is that there may be other traffic in the network during the test that affects the readings. However, as the hardware was only being used for these tests, the only other traffic was small amounts of SSH data related to the test and a small amount of traffic from system processes. The amount of this data was negligible compared to the tens of gigabytes being transferred by the transfer managers.

Two images of different sizes were used. One was a raw image with only the operating system installed that was otherwise empty. The size of this image was 9.77 GB. This image was converted to the QCOW2 format with compression to obtain the other image. The size of this image was 899 MB. The results give some indication about the benefits achieved in deployment time when images are converted to a smaller format. However, in practice images that are used for real world applications instead of benchmarking may not have as much free space and may therefore not compress this well. Using the two different image sizes should reveal if the transfer managers are able to handle images of different sizes equally efficiently compared to each other.

5.2 Disk I/O benchmarks

IOzone[55] and bonnie++[56] were used to test the disk I/O performance of a deployed virtual machine. The tests were launched inside a virtual machine running CentOS 5.5 on a physical host that was running Fedora 15. The filesystem of both the guest and the host OS was ext4. Three different deployment scenarios were tested. The first was to run the virtual machine from a raw image file, the second was to run it from a QCOW2 snapshot made out of a backing file and the third was to run it on a raw LVM partition.

The disk benchmarks work by writing to and reading from the disk in various ways and measuring the performance of these operations. Both IOzone and bonnie++ were run with settings that measure sequential disk I/O performance instead of random access performance.

The IOzone test writes a large new file on the disk and rewrites an existing file. Rewriting should be faster, as the metadata associated with the file no longer needs to be written. Read testing is done in a similar fashion: an existing file is first read and then reread. Rereading should be faster, as the file should be cached if it was previously read.[57]

The bonnie++ tests have a few differences compared to the IOzone tests. Two different methods for writing are used with two different standard C library functions: writing character by character using the putc() function and in larger blocks using the write() function. Additionally, rewriting is done to test cache performance. Similar to the write tests, read tests are done both character by character and in large blocks.[58]

To mitigate caching effects and to measure true disk I/O performance, the chosen file sizes to write must be sufficiently large[57]. The result will be inaccurate if all the data to be written fits in the memory. This will measure the speed of the memory instead of the disk. For both the bonnie++ and the IOzone tests, we chose a file size of approximately twice the available RAM: the machine used for testing had 24 GB of RAM, and the size of the test file was 47 GB. This choice was based on the IOzone documentation[57].

5.3 Task benchmarks

The purpose of these benchmarks was to measure whether the benefits gained from using the QCOW2 disk access method that would make deployments faster would be lost due to the performance of the virtual machine being lower. As a continuation to the deployment benchmarks, the specific disk access methods tested were QCOW2, raw images and logical volumes. Unlike in the I/O benchmarks, the virtual machine did not boot using a snapshot.

The QCOW2 image was used directly instead. The QCOW2 image was obtained by converting the raw image used in these tests. The image was compressed but not encrypted.

Deployments were done using the BitTorrent transfer manager for the images and the LVM transfer manager for the logical volume. However, the main purpose of this benchmark was not to test the transfer manager but the effect of the disk access method. As only a single virtual machine was deployed for each test, any transfer manager could have been chosen.

Two sets of tests were used to measure different aspects of virtual machine performance. The first set was selected to measure performance in CPU intensive tasks. The results for these should not be greatly affected by the disk access method, as the disk access method should only have an effect on disk I/O performance. The second set of tests measure performance in disk I/O. These tests should show a much larger difference between the different disk access methods. Unlike for the I/O benchmarks using IOzone and bonnie++, nothing was done to artificially limit the effectiveness of caching except to clear the disk cache on the physical host running the test virtual machine before each test run.

We selected the tests to be run from the Phoronix test suite[59]. The tests were selected so that the quantity being measured was the time to complete the test. In other words, the less time it takes to run the tests the better the result. This was done so that this number can easily be combined with the time it takes to deploy a virtual machine in order to get a single measurement for task performance.

The I/O tests selected were Flexible IO Tester[60] and SQLite[61]. Unlike the other disk I/O benchmarks that were run, these tests primarily measure random access performance. The settings for the tests were the default settings for the Phoronix test suite. The Flexible IO Tester used random reads and writes and 80% of all disk accesses were read accesses. SQLite was configured to do a lot of small writes. To simulate a long running task, the SQLite task was set to run 20 times and the Flexible IO tester 3 times. Since the run times of the two tests are very different, this ensured that both tests ran for approximately the same time.

The CPU tests selected were MINION[62] and HMMER[63]. MINION is a constraint solver and HMMER is a bioinformatics tool used for sequence analysis. As with the I/O tests, the benchmarks were configured to run multiple times to simulate an actual task. MINION was set to run twice while HMMER was set to run 30 times.

5.4 BitTorrent transfer manager optimization

We conducted further tests on the BitTorrent transfer manager to optimize its performance and to gain a better understanding of the effect the specific environment in which the deployment runs has on deployment time. There are a number of adjustable parameters for BitTorrent that affect transfer speed. For example, it is possible to adjust the number of peers the client will send data to at any one time or the total number of connections.

BitTorrent divides files to be transferred to pieces. The number of bytes in a piece must be a power of two, which gives a small number of available piece sizes. We tested all possible piece sizes from 64 kB to 128 MB.

The effect of piece size has previously been studied by Marciniak *et al.*. They concluded in their research that for small content, piece sizes as small as 16 kB are best while for larger content 256 kB is optimal[64]. They tested piece sizes from 16 kB to 2 MB. However, their tests attempted to simulate a situation where there is large variety in peer transfer capacity and all the peers have relatively slow connections. Also, the largest file size they tested was 100 MB.

In our case, all the peers have very similar transfer capacities that are also much higher than the ones tested by Marciniak *et al.* and the files are much larger than 100 MB. For this reason, we also test much larger piece sizes. The rationale behind this is that with the connection speeds available in the network, even the very large pieces can be transferred very quickly. The point where the delay from using large pieces becomes larger than the overhead associated with piece management may thus be much further in this environment.

In addition to testing different piece sizes, we tested the effect of the network topology on deployment speed. As we described in Chapter 3, there are two switches in the system which are connected using a 10 GbE link. In order to see test whether the link between the switches forms a bottleneck that slows down deployments, we deployed the same amount of virtual machines so that in one test all machines would be connected to one switch and in the other one half would be connected to one switch and the other half to the other switch. In both cases we deployed 30 virtual machines.

Chapter 6

Results and evaluation

The transfer managers we implemented were compared against two existing transfer managers: the SCP transfer manager that comes with OpenNebula and a modification to this transfer manager that replaces the SCP copy process with an NFS copy process. With the latter of these, two different NFS shares were also compared. One is located on the frontend machine ("NFS frontend") and the other is behind a storage controller connected to a separate array of disks ("NFS NetApp") as described in Chapter 3.

6.1 Deployment benchmarks

We measured two aspects of each of the transfer managers. For different numbers of virtual machine deployments, we measured both the time to complete the deployments and the total amount of data transferred to accomplish the task. The aim was to not only measure the speed of the transfer managers but also the efficiency with which they use the available resources.

Deployment speed with a large image

Figure 6.1 shows the average deployment time of a single virtual machine and the total deployment time of all virtual machines in a batch of deployments as the number of concurrent deployments grows. The size of the image for the virtual machines was 9.77 GB. The total deployment time can also be seen in Table 6.1. This is an indication of how well a transfer manager scales. As can be seen from Figure 6.1, the total and average times are very similar. In other words, the deployment finishes almost simultaneously on all the nodes. This is to be expected, as the hardware and LAN connections on the nodes are almost exactly identical.

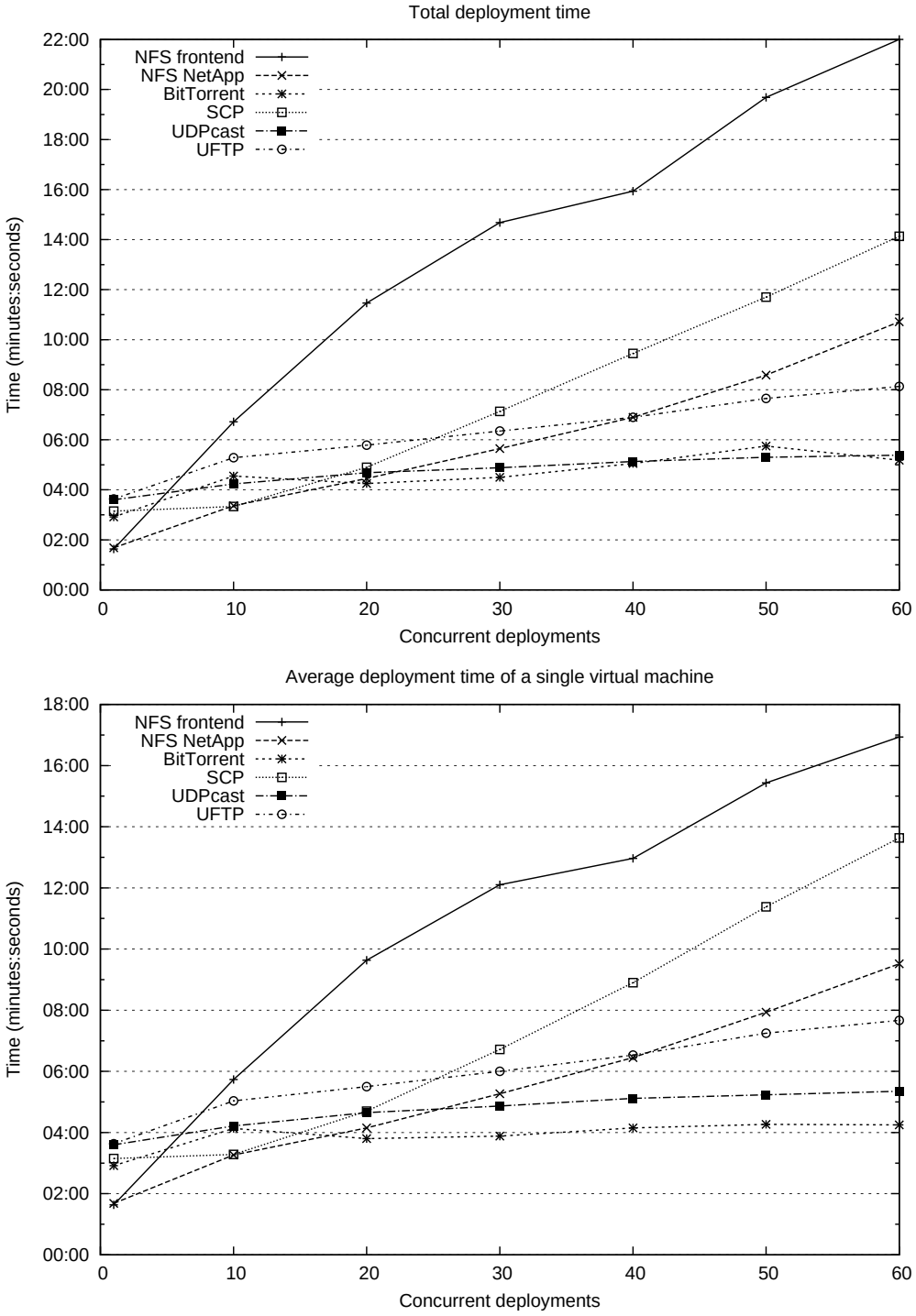

Figure 6.1: The average deployment time of a single virtual machine and the total deployment time in a batch of concurrent deployments. The size of the disk image is 9.77 GB.

VMs	NFS frontend	NFS NetApp	BitTorrent	SCP	UDPcast	UFTP
1	01:38.20	01:41.70	02:55.60	03:09.70	03:36.22	03:38.50
10	06:43.00	03:22.80	04:33.67	03:20.60	04:14.44	05:17.20
20	11:28.30	04:27.50	04:15.80	04:54.20	04:41.11	05:47.20
30	14:41.80	05:39.90	04:30.33	07:08.70	04:53.78	06:21.10
40	15:56.30	06:54.60	05:03.00	09:27.50	05:08.67	06:54.90
50	19:41.50	08:35.00	05:45.80	11:42.60	05:18.33	07:39.20
60	22:00.70	10:43.80	05:11.50	14:08.30	05:23.67	08:08.20

Table 6.1: Total time to deploy for different numbers of virtual machines using the 9.77 GB raw image averaged over 10 measurements.

One notable difference is BitTorrent. The difference between the average and the total time to deploy for BitTorrent is between 25 seconds and 1.5 minutes. This indicates that not all hosts are getting an equal share of the available resources. There are a number of reasons for why this might be happening. The algorithm that each client uses to determine which hosts to send to and which hosts to receive from may affect this. If this is the case, changing those parameters of the client software that affect this behavior – such as the maximum number of upload slots or the maximum number of connected hosts – may help.

Another issue is the piece size that has been set in the torrent file. Marciniak *et al.* have shown that this size not only affects the time it takes to transfer a file but also the variability in time between peers to finish the transfer[64]. The difference can also be related to the specific environment. Not all of the hosts have an equally fast connection to the frontend. Some are connected to the same switch as the frontend while others are connected to a different switch, which is connected with a 10 gigabyte link to the first switch. The link between the switches may become a bottleneck. We look further into these issues in Section 6.4.

Similarly to the BitTorrent case, the multicast transfer methods may also be limited by the link between the switches. However, this would not be as apparent, since the transfer session will always end at the same time for each host. In this case the issue would show up as a higher number of dropped packets on those hosts that are behind the second switch and in the case of UDPcast as the subsequent throttling of transfer speed.

When BitTorrent is used, the virtual machines on each individual host can start as soon as the image has been transferred to that host, regardless of the state of the other transfers. When multicast is used, all the hosts finish the transfer session at the same time, which means all virtual machines boot

VMs	NFS frontend	NFS NetApp	BitTorrent	SCP	UDPcast	UFTP
1	00:09.20	00:08.60	00:25.30	00:17.00	01:05.90	00:48.00
10	00:21.90	00:27.90	00:56.70	00:19.30	01:12.20	00:54.60
20	00:26.80	00:47.20	01:10.10	00:29.20	01:17.60	01:02.00
30	00:31.60	01:02.20	01:20.90	00:39.80	01:18.20	01:04.10
40	00:38.50	01:14.10	01:26.90	00:55.80	01:19.10	01:08.00
50	00:48.40	01:27.50	01:34.10	01:06.20	01:22.70	01:12.70
60	01:11.60	01:42.40	01:36.25	01:20.40	01:30.10	01:14.80

Table 6.2: Total time to deploy for different numbers of virtual machines using the 899 MB QCOW2 image averaged over 10 measurements.

at the same time. The possibility of booting individual virtual machines as soon as the transfer to the host machine for that virtual machine has finished can be seen as an advantage for the BitTorrent transfer manager. With multicast sessions, slow, unresponsive or faulty hosts can stall all the other transfers as well. There is a mechanism in both UDPcast and UFTP to drop unresponsive hosts, but it takes some time before a host can be deemed as unresponsive. Unless all hosts in a given deployment must be running before work on a task can begin, it is better to be able to use at least some of them as soon as possible.

SCP scales well up to 10 concurrent deployments. This is to be expected, as the frontend has a 10 GbE link sending data and the nodes have 1 GbE links receiving the data. Therefore, the 10 GbE link can easily serve up to 10 simultaneous transfers to the slower links. After this, the average deployment time starts to grow rapidly, as the link on the frontend becomes more and more congested. As can be seen from Figure 6.1, the approximate number of concurrent deployments where using a more efficient transfer manager than SCP is beneficial is different for each of the new transfer managers. UFTP is more efficient when there are at least 30 concurrent deployments. For UDPcast and rTorrent this number is 20.

Deployment speed with a small image

The deployment tests were repeated using a compressed QCOW2 version of the same image that was used for the first test. The results are depicted in Figure 6.2 and the total times can also be seen in Table 6.2. This time the image size was much smaller – 899 megabytes instead of 9.77 gigabytes. The high ratio of compression was due to the fact that the 9.77 gigabyte raw image was mostly empty, as it simply contained a fresh installation of

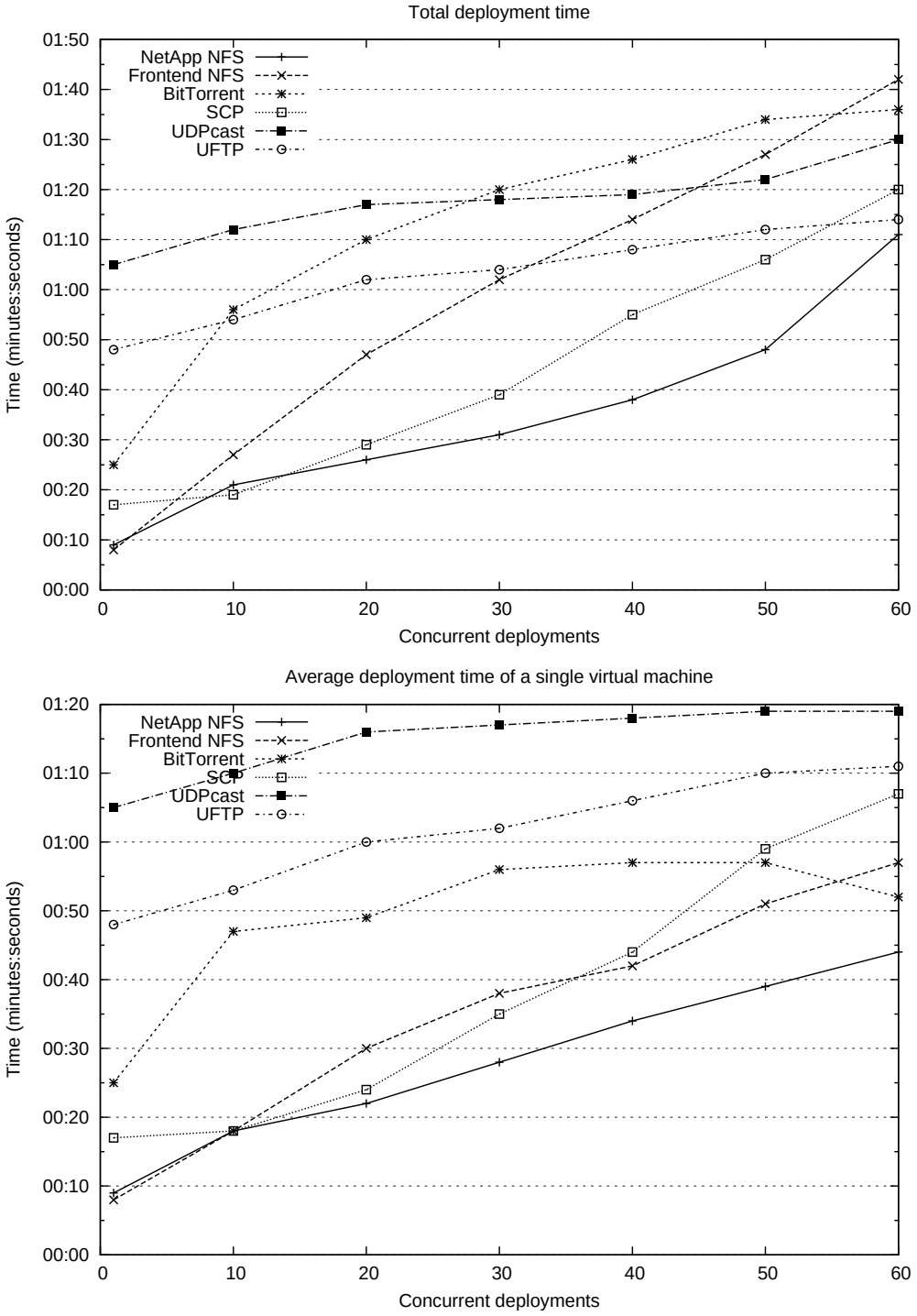

Figure 6.2: The average deployment time of a single virtual machine and the total deployment time in a batch of concurrent deployments. This test is using the image from the previous test converted to the QCOW2 format with compression. The size of the disk image is 899 MB.

CentOS 5.5 with a few additional programs installed.

It is immediately clear from Figure 6.2 that the order of the transfer managers by deployment speed has changed dramatically. The fastest option in both total and average deployment time is NFS through NetApp. UDPcast is among the slowest options in this test, even though it was one of the fastest in the other test. However, the slopes of the curves in the graph for the new transfer managers seem to be lower than the ones for SCP and NFS. While the scalability of the new transfer managers is still good, there are certain stages in them that require a certain fixed amount of time for each deployment. It seems that were there more virtual machines being deployed simultaneously, the new methods would again be the fastest. However, more measurements would be needed to confirm this.

There are a few reasons for the difference between this and the other deployment test. As the image size is smaller, the efficiency of network use is no longer the dominant factor in determining deployment time. The overhead of managing the transfer process is now more important. For example, bootstrapping a multicast transfer session takes more time than bootstrapping a unicast session. With multicast, the server must negotiate with and wait for a number of hosts instead of just one. The transfer can only begin when all hosts have either replied or timed out, whereas individual unicast transfers can be started as soon the client has replied.

The UDPcast and UFTP transfer managers spend a significant amount of time on tasks that are necessary to form the multicast groups. The instance of the transfer manager that starts the server must receive information from all the other instances about which locations the image should be transferred to. This master instance must wait before starting the transfer so that all instances have time to communicate the location where they want the image for deployment. Various delays and uncertainties mean that the time to wait must be relatively long. For example, not all the transfer manager scripts are guaranteed to start at the same time due to the OpenNebula scheduler and some of them may be delayed while writing to the shared files.

In addition to the coordination overhead, the UFTP and BitTorrent transfer managers must first transfer the image to a temporary location on each host and then copy it from that location to a final location for deployment. This is necessary as only one transfer will run per host. The additional copy operation is faster than the transfer, but it still takes a long time. Possible solutions to these issues are discussed in Chapter 7.

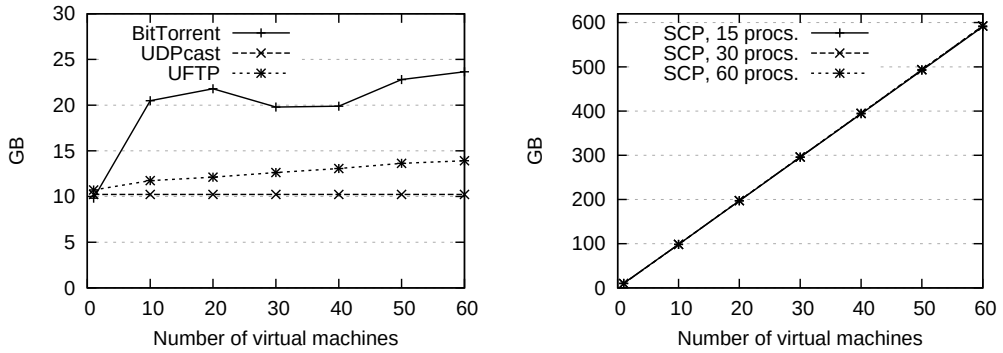

Figure 6.3: The total amount of data transferred from the frontend machine that is serving the image to the virtual machine hosts, with different numbers of deployed virtual machines. Note that SCP is presented in its own graph due to a large difference in scale.

Amount of data transferred

Figure 6.3 shows the increase in data transfer from the frontend machine as more virtual machines are deployed. Note that SCP was separated into its own graph since it generates far more traffic on the frontend than any of the other transfer managers. The size of the virtual machine image transferred in this test was 9.77 GB. In the ideal case, only this amount of data would be transferred regardless of how many virtual machines are being deployed and there would be no overhead caused by, for example, retransmission or informational messages related to the protocol. In practice, there will always be some overhead. One measure of how well a transfer manager scales to a large number of deployments is the amount of overhead information transferred during deployments. If the overhead of a given transfer manager is small and grows very slowly as the number of virtual machines increases, then that transfer manager most likely scales well.

The traffic generated by the default SCP transfer manager was measured for comparison purposes. As the measurements were made using the same OpenNebula scheduler settings for all transfer managers, this meant that during the test the frontend machine was running as many concurrent SCP transfers as there were machines being deployed. In other words, when 60 virtual machines were being deployed, there were 60 concurrent SCP transfers. In practice, it would be better to limit the number of concurrent deployments to limit the load on the system. However, brief testing with different parameters seemed to indicate that the fastest way to get all the virtual machines

running is in fact to run all the transfers concurrently, though this is mostly likely not true for much more than 60 virtual machines. The optimal number of concurrent transfers for SCP was not investigated further.

During preliminary testing, the amount of data transferred when using the SCP transfer manager scaled exponentially as more virtual machines were deployed and OpenNebula was set to concurrently deploy a maximum of 60 virtual machines. For 60 virtual machines, the amount of data transferred was more than 1 TB. This was most likely due to the load caused by so many concurrent transfers. The send and receive buffers on the system got filled faster than packets can be processed, which caused packet drops and subsequently retransmissions. For this reason, we also chose to measure the amount transferred when using 15 or 30 simultaneous deployments. When the number of concurrent deployments was limited to either 15 or 30 in the scheduler, the amount of data grew linearly as expected. However, we did not observe the same exponential growth with a limit of 60 processes in the final test where 10 separate measurements were averaged.

As is to be expected, the multicast based UDPcast and UFTP transfer managers produce the least amount of traffic on the frontend. UDPcast is the most efficient, as its overhead remains almost constant as the number of deployed virtual machines increases. The overhead of UFTP increases steadily, and with 60 virtual machines deployed, about 30% of the data transferred is overhead. With UDPcast the overhead is approximately 6,7% regardless of the number of nodes.

The difference between the two is most likely due to their different rate limiting mechanisms. UFTP must be set to always transfer at a specific rate, whereas UDPcast can dynamically vary the transmission rate to achieve a more optimal transfer rate. As the number of receivers grows, the likelihood that at least one receiver misses a given packet increases. The likelihood of missed packets also increases the faster the transmission rate. Using a single transmission rate while avoiding dropped packets as much as possible means that either the transmission rate is too slow if there are few receivers or too high if there are many receivers.

The total amount of data transferred in the whole network when using BitTorrent is far larger than with the multicast transfer managers and should be close to the optimal case of data transferred when using SCP. In other words, it is approximately the number of virtual machines times the size of the virtual machine image. However, BitTorrent distributes the transfer to multiple physical hosts, which means only a very small part of the data is transferred by the frontend. The amount of data transferred from the frontend when using BitTorrent shown in Figure 6.3 grows slightly as the number of deployed virtual machines grows, but not nearly as much as it

does with SCP.

One interesting thing to note about the amounts of data transferred is that even though with BitTorrent more data is being transferred from the frontend, BitTorrent is still as fast or faster than the multicast options. This is again because of the 10 GbE link serving the 1 GbE links on the hosts. The multicast transfer managers only use a very small part of the available bandwidth, as the speed is limited by the 1 GbE links on the hosts. While with 60 deployed virtual machines the frontend is not yet a bottleneck with BitTorrent, if the amount of data that needs to be transferred continues to grow, this may start to negatively affect the deployment time.

6.2 Disk I/O benchmarks

Compressed QCOW2 images make virtual machine deployments faster as they are smaller than raw images. However, a machine deployed using a QCOW2 image has lower disk I/O performance than a machine that uses either a raw disk image or an LVM partition. We ran a set of tests with different QEMU cache settings and disk access methods to see how large this difference is. If the difference is small enough, then it is beneficial to use QCOW2 images because they are fast to deploy. On the other hand, if there is a large gap in performance, then the benefit gained from a shorter deployment time will be lost due to tasks on the virtual machine taking longer to run. How large the gap in performance can be depends on the task that is to run on the virtual machine. If the task is disk I/O intensive, then even slightly reduced disk I/O performance can make a large difference in task run time. If the task is CPU intensive and uses almost no disk I/O, then even a large gap in disk I/O performance can be justifiable if the deployment time can be reduced.

The results can be found in Figure 6.4 and Table 6.3. In the IOzone benchmark, LVM is the fastest disk access method overall. This is to be expected, as it is the method that has the least amount of overhead. Unlike QCOW2 and raw images, only one filesystem is involved, and disk accesses are more direct. Read and write performance are approximately equal. In the IOzone benchmark, all the different cache settings provide more or less equal performance, whereas in the bonnie++ benchmark there are some differences. These are most likely due to the different methods used for reading and writing compared to IOzone. Overall, LVM has the best performance in both benchmarks. It serves as an indication of the maximum sequential disk I/O performance that can be expected from a virtualized environment.

Both image formats incur a performance penalty compared to LVM. This

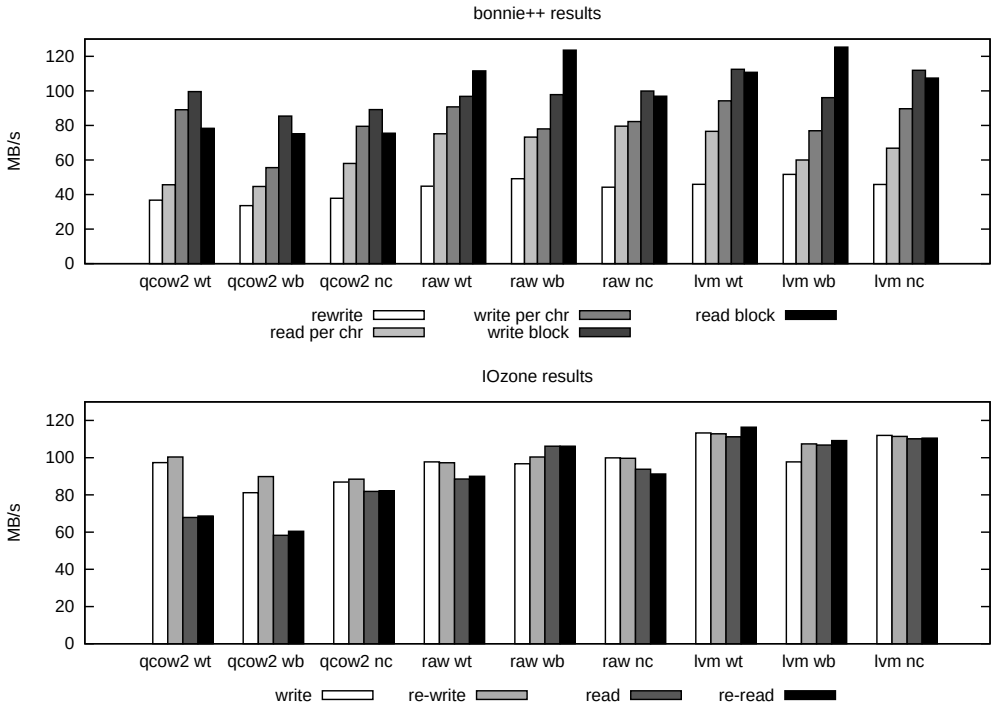

Figure 6.4: The performance of different disk formats and different KVM cache settings (WT = writethrough, WB = writeback, NC = no cache) as measured by bonnie++ and IOZone.

		QCOW2/LVM	QCOW2/raw	Raw/LVM
IOZone	write	85.94	99.61	86.28
	re-write	88.99	103.21	86.22
	read	61.06	76.70	79.61
	re-read	58.99	76.29	77.33
bonnie++	write per chr	94.53	98.12	96.34
	write block	88.49	102.80	86.09
	rewrite	79.93	81.99	97.48
	read per chr	59.62	60.74	98.15
	read block	70.73	70.21	100.74

Table 6.3: The disk I/O throughput of a QCOW2 image in percentages relative to raw images and LVM and the throughput of raw images relative to LVM. These numbers are based on the benchmarks in Figure 6.4. Writethrough caching was used.

is to be expected, as there is an additional file system layer. QCOW2 images incur an additional performance penalty compared to raw images. The use of compression and the need to grow the images as data gets written are possible reasons for this penalty.

In the bonnie++ results, per character read performance is significantly worse for QCOW2 when caching is used. This is also the case for the read performance in IOzone. However, the block read performance in bonnie++ is not affected by the chosen caching method. This may be an artifact of how the tests work. Data is first written and then that same data is read. As the data is written, it is also cached. There is approximately twice as much data to be written than there is memory available for caching. This means only half the data will fit in the cache.

Cached reads will always check the cache first, but because only half of the data being read is in the cache, there will be a lot of cache misses. On the other hand, when no cache is used, half of the reads will be unnecessarily slow but the other half will not get slowed down by a cache miss. The reason why the block reads in bonnie++ show different results may be that the caching system can in their case predict future reads better and act accordingly, thus leading to less cache misses. When reading character by character, the information about what will be read next is not available.

The reason raw images or LVM are not affected in the same way could be that cache misses are less expensive in them. Cache misses should not be expensive compared to the time it takes to read data from a disk, as the RAM used for caching is two orders of magnitude faster than the disk used for storage. If they are expensive, this means there is more overhead than necessary in determining whether a given piece of data is in the cache or not.

One thing that is different in QCOW2 from the other formats in this respect is its two-level lookups necessary for locating a specific piece of data on the disk: the image is divided into *clusters* which are further subdivided into *sectors*[30, 31]. To find a specific piece of data, the correct cluster is located first and then the correct sector inside that cluster. This may be the reason for poor cache performance in this benchmark.

Based on these results, it would seem the best option for QCOW2 is to not use caching. However, both IOzone and bonnie++ are synthetic benchmarks. Real applications may exhibit different behavior when caching is used. Further testing would be required to determine the best caching method.

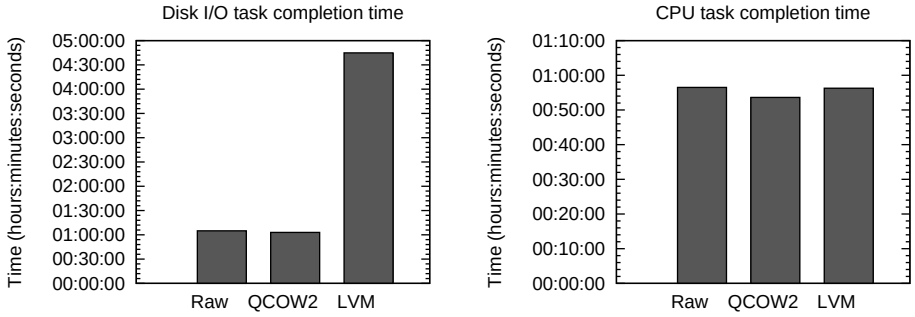

Figure 6.5: The time from deployment start to finishing a given benchmarking task with two different image formats. The measurements were repeated 5 times and the results averaged.

6.3 Task benchmarks

The results for the task benchmarks that comprises deployment time combined with repeated runs of Flexile IO Tester and SQLite for disk I/O and MINION and HMMER for CPU can be found in Figure 6.5. The disk I/O part of these tests is similar to the I/O benchmarks except for the type of I/O load used. Instead of using sequential reads and writes, random reads and writes are used instead. This means that the effect of access latency is more pronounced than sequential read or write speed.

For the type of I/O access pattern used here, the QCOW2 image when used as is performs very well compared to the raw image. The performance is practically the same, and the small advantage that QCOW2 has here comes from the fact that the image was deployed faster as it is smaller. The CPU tasks show a similar difference in performance, again caused by the QCOW2 image being deployed in less time. LVM performance is identical to the image formats in the CPU tasks.

The most surprising result is the poor performance of LVM in the disk I/O benchmark. As there should be less software layers between the virtual machine and the hardware device when using LVM, disk access time should be shorter instead of longer. Instead, running the benchmark takes almost five times as much time as it does with either image based disk access method. There are several possible explanations.

The virtual machines which were used to run the task benchmarks were set to have 4 GB of memory. However, the physical host has 24 GB of memory. Even though only 4 GB should be available, it is possible that the

host operating system is using the rest of the memory on the physical host to cache data from the virtual machine. This caching effect in turn might not be identical for all the different disk access methods. To make sure this is not the case, we ran the LVM test again on a virtual machine that had all of the memory of the physical host available to it. The performance of the virtual machine did not improve.

As the tests are repeated several times in a single run, it is possible that the data written by the previous test runs and stored in the disk cache is affecting the results. If for some reason the machine deployed using LVM could not use the disk cache as effectively as the other options, this could explain the performance difference. Another possibility is that the disk caching is negatively affecting performance. We ran the LVM tests without any disk caching on the virtual machine to make sure this is not the case, but performance did not improve.

Given the tests that we performed, it seems there is an implementation or a parameter problem in one of the software components involved in disk accesses for LVM volumes used by virtual machines. As we have discussed in Chapter 1, from a theoretical point of view, performance should be much better.

The only difference using QCOW2 images make in these tests is that they can be deployed faster and the tasks finish sooner. Benchmark performance does not seem to be affected at all. However, these tests are not a worst case scenario for QCOW2. The compression used in QCOW2 is read-only, which means any new block written while the virtual machine is running will be written uncompressed[65]. The I/O tests performed here write new data and read that instead of reading data already on the image. The operation is not very different from the raw image case. However, one difference is that as new data gets written, the QCOW2 image must grow. This incurs some overhead, but if each write operation only writes a small amount of data, access latency affects the result more than this overhead. From the point of view of this benchmark, filesystem performance is therefore very similar to what it is when raw images are used. Because they write more data, the sequential disk I/O benchmarks described previously are more difficult for QCOW2. The performance of QCOW2 is accordingly lower in those benchmarks.

6.4 BitTorrent transfer manager optimization

Based on the encouraging results for BitTorrent in the deployment benchmarks and experiences gained on stability and maintainability of the transfer

manager scripts, we decided to further explore the optimal parameters for BitTorrent transfers.

The effect of piece size on deployment time

Figure 6.6 shows the average per machine deployment time and the total deployment time of 60 virtual machines using different BitTorrent piece sizes. The same 9.77 GB image that was used in the deployment tests was used here as well. The optimal piece size when average deployment time is considered is 256 kB, which is in line with the results of Marciniak *et al.* for large files[64]. However, the differences are not very large for reasonably sized pieces. For example, the difference in deployment time between 256 kB and 8 MB is only about 5 %. It is only when the piece size is much larger that the deployment time is significantly slower. The variation in average completion time also appears to be smallest with those piece sizes that are closest to the optimal 256 kB.

The most interesting result is that the total deployment time does not follow the same pattern with respect to piece sizes as the average deployment time. Up to 8 MB, piece size appears to have almost no effect on the time when the last of the transfers is finished. Interestingly, 8 MB is an exception and seems to yield the fastest total deployment times. We do not know the reason for this, and further study would be required to rule out any measurement error. In any case, the difference to the smaller piece sizes is still quite small, especially considering the high degree of variation in the results for a given piece size.

It appears there is a phenomenon separate from piece size that is affecting the total deployment time. The endgame algorithm used by clients that have almost finished a download and are only missing a few pieces could affect this. In endgame mode, clients send requests for missing pieces to all other clients. As pieces are received, pending requests for those pieces are canceled[12]. As all hosts have the same connection speed and start the download almost simultaneously, a large part of the clients are also likely to enter endgame mode simultaneously. This will generate a lot of requests and the amount of overhead will grow as most of these requests will be unnecessary.

As the clients finish the download, they will start a hash check of the image. They will not answer requests while the hash check is in progress. Similar to the endgame mode, many clients will start the hash check almost simultaneously, leaving them unresponsive to requests. When this is combined with many clients being in endgame mode and sending a lot of requests, it is likely that the last few clients to finish are left to wait for some time before they get the final few pieces.

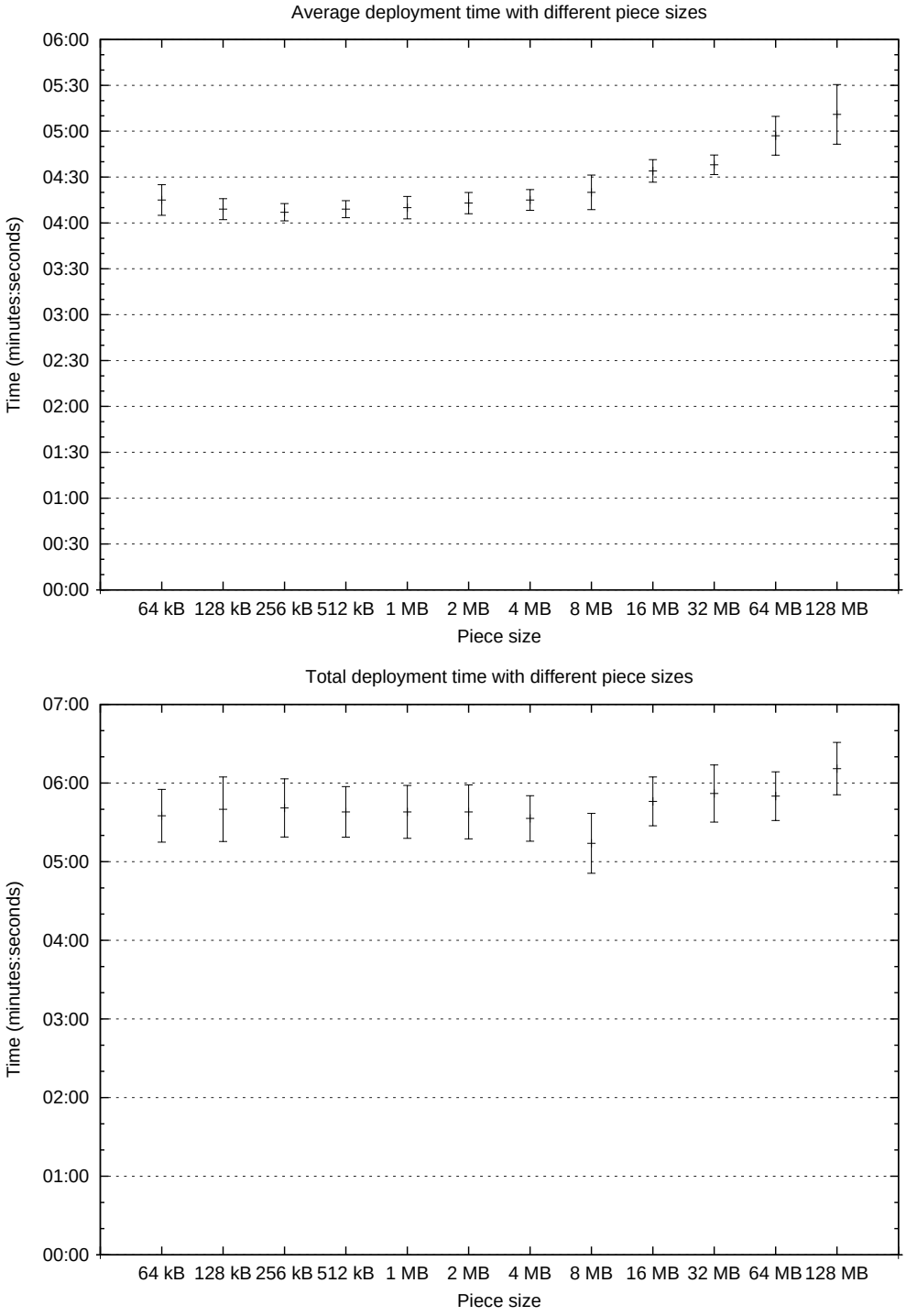

Figure 6.6: The effect of torrent piece size on total deployment time. For each piece size, the time to deploy 60 virtual machines was measured 20 times and the results averaged. The averaged result for each piece size is presented along with one standard deviation in either direction. The image size was 9.77 GB.

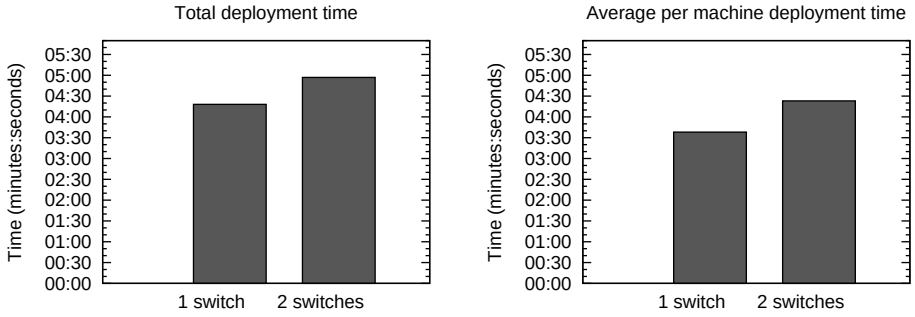

Figure 6.7: The effect of using two switches instead of one for 30 deployments. For the single switch test, all 30 virtual machines were deployed to hosts that were connected to the same switch as the frontend. For the test with two switches, 15 hosts were connected to the same switch and 15 to the other switch in the system. The test was repeated 20 times for both scenarios and the results averaged. The piece size was 4 MB.

The purpose of the endgame mode is to prevent clients from getting stuck by requesting the last few pieces from unresponsive or slow clients. The purpose of the hash check is to ensure data integrity in an environment with potentially malicious or malfunctioning hosts and unreliable network connections. These problems are rare in the type of environment used in this thesis. The endgame mode and hash checking are therefore mostly counterproductive. However, removing the endgame mode completely could in some instances lead to very slow deployment times. The best approach would most likely be to modify it so that instead of sending requests to all the clients, the last few pieces are instead requested from a small number of clients that are preferably spread throughout the network.

The effect of peer closeness on deployment time

Figure 6.7 shows the effect the link between the two switches has on deployment speed. The same amount of virtual machines was deployed using physical machines connected to only one of the switches and physical machines connected to both switches. The difference is on average about 40 seconds or about 15 %. We believe there would be a bigger difference in a larger system with more switches and connections between them, as packets would traverse more than one of these crosslinks between switches. As was discussed in Chapter 2, steps should be taken to reduce the traffic on the crosslinks in order to maximize performance. In practice, this means

selecting some heuristic to determine the closeness of peers from a network topology point of view and then biasing the closest peers in peer selection.

Various heuristics for determining peer closeness in the Internet were discussed in Chapter 2. It is easy to create a heuristic for peer closeness for the test system used in this thesis. All the machines belong to the same subnet, but the switch a machine is connected to can be reliably determined based on the third octet of the machine's IP. If a similar IP address scheme is followed, it will be simple to create heuristics for peer closeness even in much larger datacenter environments. For example, a vector can be formed from the IP address and the distance of two IP addresses calculated as the cosine similarity between their corresponding vectors similarly to what is done by Choffnes and Bustamante[49]. Weights need to be added to the vector components so that the least significant bytes do not add as much distance as the more significant bytes. If only the least significant byte changes between two IP addresses, the machines are connected to the same switch and their distance can be assumed to be zero.

Chapter 7

Discussion

7.1 Complexity of setup and usage

While the speed and the scalability of a transfer manager are its most important characteristics, other factors can also affect its usefulness. One of these is the complexity of setting up, using and maintaining the transfer manager. The complexity in the transfer managers comes from different sources. The complexity of the transfer manager script varies, but even if it is simple, the other related infrastructure can be complex.

All of the transfer managers require one or more external processes to handle the actual transfer. These processes may only be launched as they are needed, or they may run constantly. The more components there are in a transfer manager the more likely it is for one these components to fail. Software that needs to run constantly on virtual machine hosts for the transfer manager to operate correctly will consume resources that could be used by the virtual machines to do actual work. Security and other updates will need to applied to the software. The larger number of running software will also mean that the attack surface of the host grows.

The least complex transfer managers are the shared and SSH transfer managers that are included in the OpenNebula distribution by default. They use very few commands to clone virtual machines and require no additional software besides what is most likely already installed. From the point of view of additional infrastructure required, the UDPcast transfer manager comes closest to their simplicity. The only software needed that is not installed by default on the virtual machine hosts and the frontend machine is UDPcast and no constantly running software is needed, as the scripts of the transfer manager start and tear down any necessary processes. However, the cloning script of the transfer manager is more complex than the simpler default

scripts. It uses locks and shared files, which increases its complexity and the probability of bugs appearing, but if everything works correctly, the transfer manager is more transparent to the user than, for example, the BitTorrent transfer manager.

The UFTP transfer manager is similar to the UDPcast transfer manager. The logic used to manage the transfer to a certain subset of the available hosts is very similar. One difference is that instead of only using processes launched by the script, the transfer manager relies on a UFTP daemon to be constantly listening on the target hosts. Another difference is that there is an additional step where the image is copied from a temporary location on the target host to the final location for deployment. This means that these temporary images need to be cleaned up periodically, which introduces more complexity.

The complexity of the BitTorrent transfer manager comes from a different source. The cloning script is quite simple, and unlike the UDPcast and UFTP transfer managers, no concurrent programming is involved. However, it requires an rTorrent process to constantly run on the server and the virtual machine hosts, an HTTP server on all the virtual machine hosts to forward XML-RPC commands and a tracker for tracking connections. All of these processes consume resources, and the resource consumption of the HTTP server and the rTorrent client are of most concern, as these run on the virtual machine hosts that are the hosts that do the actual work in the cloud system. Furthermore, additional torrent files need to be created and maintained for all the images in the repository. While a torrent file will need to be generated only once for each new image, generating the files can take a long time when image sizes are tens of gigabytes.

The parameters of each transfer manager required a different amount of tuning. BitTorrent worked well with any settings that were used. As it is TCP based, the reliability is provided by the TCP implementation of the operating system. UFTP and UDPcast are instead based on UDP, which means they must have their own mechanism for reliability. We found out during development that these mechanisms are such that care must be taken to select parameters that provide reliable transfers. Choosing the wrong parameters will lead to dropped packets which will cause unnecessary delays and traffic and may lead to transfers timing out. While the parameters may be correct for one situation, it is difficult to make sure that they are correct in others. For example, increased traffic may create problems if congestion is not handled properly.

7.2 Transfer method suitability

The image transfer methods that we investigate in this thesis are not specifically designed for virtual machine image transfers in a controlled data center environment. How close the intended use case is to this particular task depends on the transfer manager. For example, UDPcast is intended to be used, among other things, for installing operating systems on a large number of physical machines[16]. Multicast is well suited for this task as the fact that operating systems are being installed implies that the environment is fully controlled by the system administrator.

On the other hand, BitTorrent is designed for a very different environment, where users have a wide variety of different hardware and connection types. Users may also choose to run their own BitTorrent software, and not all software that connects to the BitTorrent swarm can be assumed to be fair or truthful in reporting its status to the swarm. Two examples of BitTorrent clients that attempt to exploit weaknesses in the BitTorrent protocol to gain a disproportionate amount of the available resources are BitTyrant[66] and BitThief[67].

The issues of connection, hardware and client variety or attempts at freeloading do not exist in a controlled data center environment. It is true that there may be some variability in hardware and network capabilities. The basic system may use older hardware than a newer extension to the same system. Some hosts can be connected using 1 GbE connections while others use 10 GbE connections. However, there are generally only a few different levels of performance and not the tens or hundreds that can be expected in a public BitTorrent swarm.

7.3 Future software support

One practical issue to consider when selecting a transfer manager for production use is whether the software it requires is actively developed and either has features that will be critical in the future, such as IPv6 support, or will have these features in a future version.

Ideally, the transfer manager should not rely on a specific piece of software. There should be a number of different suitable options for the programs required by the transfer manager, and the programs used should be easily replaceable without major code changes. If this is not the case, then there is a risk that in addition to the transfer manager scripts, the original software required will also need to be maintained if the original author stops maintaining it. Each of the programs used in the new transfer managers is

maintained by only one developer. This means there may be a higher than usual risk that these programs stop being maintained.

All of the software used by the transfer managers we have evaluated appears to be actively maintained[14–17, 51]. However, there are some differences in the replaceability of the software components. The interfaces of UDPcast and UFTP and the way they are used by the transfer manager scripts means that if for some reason these programs could no longer be used as they are, replacing them would require a considerable development effort. As far as we know, there are very few available reliable multicast file transfer programs other than UDPcast and UFTP.

While an unmaintained program will still function, new features will not be added and security issues will not be fixed. The IPv6 support we mentioned is a good example of a feature that is not needed now but may be critical in the future. As far as we can tell, this support is not found in the default versions of UDPcast, UFTP or rTorrent[14, 16, 17]. However, there are more alternatives to rTorrent than there are to either UDPcast or UFTP. One example is Transmission, which supports IPv6[68]. There are also multiple pieces of tracker software that can be used.

7.4 Potential improvements

The biggest improvements can be made through modifications to the transfer manager algorithms. As was discussed in Chapter 6, the coordination process takes a disproportionately large amount of time. By using better locking mechanisms and reducing the number of locks in a controlled way, the delays could be greatly reduced. However, the general architectural model for transfer managers in OpenNebula is not optimal for multideployments. Changes to accommodate multideployments would make the deployment process not only more efficient but also make the deployment code easier to write and maintain.

Improvements to the current algorithms

The current lock implementation used by the multicast transfer managers is based on using the standard Unix command *mkdir* with an absolute path name, which is an atomic operation[54]. This method has many shortcomings. There is no good mechanism to wait on a lock. Waiting is instead based on polling, and there is no queue for waiting processes. This is potentially unsafe and may also lead to starvation.

As the transfer managers are written in bash, there is only a limited number of options to choose from for locking. A better option than using mkdir would be to use the Linux program *flock*[69], which is a part of the *util-linux*[70] set of tools. It uses the system call *flock()*[71] for locking. This would make the code safer and more efficient, as there would be no polling involved.

The BitTorrent transfer manager does not use locks at all and the transfer manager is much simpler in other ways as well, as most of the work is handled by the torrent clients and the tracker. There is one instance of polling, where the script checks for image completion on the target host, but the delay introduced by this is small compared to the total transfer time. The checks also generate some network traffic as they must be done from the frontend to the target hosts, but the amount is very small compared to the image data that must be transferred. A better solution would be to have the script that runs on the target host when a download finishes send a notification to the transfer manager script on the frontend. The transfer manager script could then be suspended while it waits for the notification.

A more involved method of optimization would be to modify the transfer software to better suit the task of image transfers. One example is to modify rTorrent so that it can write data directly to a logical volume, read that data and seed it to other peers in the swarm. This would make it possible to write data arriving from the network directly to an LVM volume and once the transfer is finished, create a snapshot of that volume to boot from and continue seeding from the original logical volume. The client could also switch seeding to the snapshot and then boot from the original LVM volume to avoid the performance penalty of using an LVM snapshot[11] for running the virtual machine.

OpenNebula transfer manager model shortcomings

The current OpenNebula transfer manager model is tailored to deploying one virtual machine at a time. The deployment process of a virtual machine always involves running the same cloning script that handles copying the file to a physical host. This solution works well when only a few virtual machines would ever need to be deployed simultaneously and no coordination between the scripts is needed. In other words, the current transfer manager system is suited to multiple independent unicast transfers. However, certain transfer methods – such as multicast – treat the transfer of one image to multiple physical hosts as a single transfer session.

Having to coordinate instances of the same script so that a single session is successfully established and the image file is transferred to all the neces-

sary locations is overly complicated and prone to errors. This means that implementing transfer managers is more time consuming and the end result is more difficult to maintain. We propose a more centralized approach, where one process can deploy multiple virtual machines at once. This process would take as input an image name, a list of hostnames and a list of locations on those hosts for the image. There would be no need for inter-process communication or concurrent programming. The transfer manager implementation would require less code and would be easier to maintain and extend.

The logic of OpenNebula would have to be changed so that instead of calling multiple scripts, it would instead provide one process with all the necessary information for completing the deployment to multiple physical hosts. Working around the problem once the transfer manager script has already been called is not possible, as no single instance of the script has all the necessary knowledge. Instead, the script instances all have one small part of the necessary information that they must share with the others, which leads to the coordination problem.

While improvements in the logic of the transfer managers we have implemented could significantly reduce deployment time, these improvements would also introduce additional complexity. A better solution would be to solve the original architectural problem instead of working around it.

Optimization of the local copy in BitTorrent and UFTP

As we have previously discussed, having to copy data separately to one or more locations for deployment after the transfer is finished slows the deployment process significantly. In the case of UFTP, the transfer manager could be modified so that it works the same way as the UDPcast transfer manager and writes the data simultaneously to the final locations. The BitTorrent transfer manager can not do this, as the original image must be preserved for uploading to the other physical hosts so that the deployments on those hosts can be made faster as well.

There are various implementations of the copy-on-write pattern that can be used to reduce the time required to create the copy to nearly zero. One of these is using QCOW2 in backing file mode, but there are also filesystems that have similar functionality. By using one of these filesystems, the user is not tied to any specific image format.

During initial development and testing of the transfer managers, we experimented with Btrfs[72], which supports copy-on-write copies. The operating system used was Fedora 15. The stability and performance of the filesystem was not sufficient at the time, but as Btrfs is still under heavy development[72], these aspects have most likely improved.

Another filesystem with this feature is ZFS[73], but a stable native version of it is only available for Solaris. There is a separate version in development for Linux, but due to licensing issues it will not be included in the Linux kernel[74].

If performance and stability issues can be solved, these and other similar filesystems can be used in the future to provide a significant improvement in deployment time. If combined with a fast network share they can be used to provide nearly instant deployments in a similar way to what is described in Section 7.8.

7.5 Summary of the transfer methods

The transfer managers that we implemented are summarized in Table 7.1. All of the implemented transfer managers provide better performance for large concurrent deployments than the default options when the image to be deployed is sufficiently large. For single deployments, small deployments and small image files, the performance of the new transfer managers is slightly worse than the performance of the default transfer managers. Further development and optimization should mitigate these problems by reducing the overhead associated with managing the transfer process.

One potential problem in each of the transfer managers is how the software that they use will be supported in the future. However, it is possible to use unmaintained software as long as there are no outstanding issues or necessary feature additions in the near future. BitTorrent is the best alternative from a software support point of view, as there are alternatives available for each of the components.

Transfer script complexity is something that is partly a product of the transfer manager architecture of OpenNebula. If all the information necessary for a large deployment could be provided in one place, there would be far fewer implementation issues. Out of the implemented transfer managers, this issue affects UDPcast and UFTP. They must rely on locks and shared files for coordination between multiple transfer manager scripts.

The BitTorrent cloning script is simple, but there are other sources of complexity. For each image, a torrent file must be created and kept up to date. Separate tracker software must be used. Two image copies exist on the host and only one of these will be deleted when the virtual machine is deleted, as the other one may still be in use by the torrent software. One must make sure the other copies are deleted before disk space runs out.

BitTorrent ensures reliable transfers by using TCP, which means it benefits from the well tested TCP implementation of the operating system.

	New TMs			OpenNebula default TMs	
	BitTorrent	UDPcast	UFTP	SCP	NFS
Single deployment speed	Good	Good	Good	Good	Very good
Multideployment speed	Very good	Very good	Good	Poor	Poor
Additional local copy operation	Yes	No	Yes	No	No
Transport layer protocol	TCP	UDP	UDP	TCP	TCP
Number of developers	One[a]	One	One	Multiple	Multiple
Software source code availability	Yes	Yes	Yes	Yes	Yes
Transfer manager script complexity	Simple	Complex	Complex	Simple	Simple
Software environment complexity	Complex	Simple	Simple	Simple	Simple
Requires constantly running processes	Yes	No	Yes	No	No
Parameter fine tuning necessary	No	Yes	Yes	No	No
IPv6 support	No[b]	No	No	Yes	Yes

Table 7.1: A summary of the transfer manager features

[a]There are numerous alternatives to the client and tracker software used for this thesis.
[b]Other BitTorrent clients with IPv6 support are available.

UDPcast and UFTP use UDP, which means they must have their own mechanism for reliability. These mechanisms are not as well tested, and they may have unforeseen issues. Increased traffic in the network used for the deployment may be one possible cause of problems. The mechanisms must also be adjusted by the user to provide reliable transfers, which leads to increased management and the possibility of selecting parameters that work well in one situation but do not work in another.

As far as we could determine, none of the three programs used for transfers in the implemented transfer managers currently support IPv6. This may be an issue in the future as IPv6 replaces IPv4. However, the internal network used for deployments will be able to use IPv4 for a long time. If deployments involving remote image repositories are needed at some point, then the lack of IPv6 support may be a problem sooner. There are also alternatives to rTorrent that support IPv6, such as Transmission.

7.6 Summary of the disk access methods

None of the disk access methods is best in all situations. According to the benchmarks we ran, LVM is best for sequential access, but has problems in at least some situations where random access disk I/O is required. There are also other considerations besides performance. QCOW2 images are smaller than raw images, so they require less disk space for storage and can be deployed faster. QCOW2 snapshots can also be used to speed up deployments.

The choice of the caching method also depends on the situation and the disk access method used. According to our benchmarks, no caching should be used with QCOW2. Writeback caching provides the best performance for raw images and writethrough caching for LVM. However, these results do not apply to all possible tasks involving disk I/O. Further study is required to get a better picture of the performance in different situations.

7.7 Usage recommendations

The choice of the transfer method depends on the usage model of the cloud environment. One model is to have virtual machines that are generally long running, are used to perform several tasks in their lifetime and only few are ever deployed simultaneously. This usage model is not very different from using physical servers and is not very flexible. For this model, the transfer manager that provides the best performance for single deployments should be chosen. Out of the transfer methods evaluated here, copying images over

NFS is the best choice for this model. As we have shown through evaluation, it is faster than the other alternatives for single deployments due to having little or no overhead related to, for example, transfer group management.

More efficient transfer methods make new usage models viable. One is to deploy one or more virtual machines to perform a single task and then tear them down once the task is finished. This is only possible if deploying new virtual machines is fast. If it is not, then the size of the tasks will be limited by how many virtual machines can be deployed to work on a task. We have shown through evaluation that efficient transfer methods provide significantly better scalability than the default transfer options of OpenNebula even in the small test environment. Out of the evaluated transfer managers, we recommend BitTorrent as the best option for this usage model. Its performance is similar to UDPcast, but it avoids some of the problems related to implementation using the current OpenNebula transfer manager model.

The choice of disk access method depends on the task that will run on the virtual machine. If the task involves little or no disk I/O, then QCOW2 is the best choice due to its small size and other features, such as the ability to create snapshots to speed up the deployment process. QCOW2 is also good for tasks that only involve random access disk I/O. LVM should be used if there are a lot of sequential accesses, but as we have shown in the task benchmarks, it does not provide better I/O performance in all situations.

7.8 Other potential deployment methods for OpenNebula

Some deployment methods were not tested due to limitations of available time and resources. Two of these methods and their possible merits and disadvantages are mentioned here to provide information for further studies. Both of these methods allow virtual machines to be deployed nearly instantly, but there may be a disk I/O performance penalty. Using storage through the network to avoid costly bulk transfers of data is the key idea in both methods: this idea was already discussed in Chapters 1 and 2, but here we provide a more practical look.

iSCSI volume snapshots

iSCSI is a protocol that allows communicating with I/O devices through IP networks using SCSI protocols[75]. The SCSI commands are carried on top of TCP. It can be used to provide access to a block device that appears to be

local but is instead located on a remote machine. If the block device represents a virtual machine image, a virtual machine can be deployed using it by making a snapshot of the block device. The information on the image will then be accessed through the network, and because a snapshot was made, the original block device will maintain its state. As snapshots can be created almost instantaneously, this method provides nearly instant deployment times. The snapshot can be provided through either a proprietary mechanism or by using LVM. Implementing this setup in practice would be simple. The snapshot can be created with a single command that is run inside the transfer manager script.

With this method, the network between the block devices and the physical hosts where the virtual machines run must be very fast so that it does not become a bottleneck. The image repository must be able to serve multiple concurrent I/O requests from multiple virtual machines without a significant drop in performance. In practice, this means that multiple disks and physical network connections must be used in parallel to provide sufficient throughput. The more running virtual machines there are, the more performance must be centrally available on the image repository. This means that the hardware required may be prohibitively expensive at least for large cloud systems.

While this type of an implementation allows for fast deployment times, it does not fully use the distributed performance available through the disks already on the nodes of the cluster. Instead, much of the load is placed on the image repository which must accordingly have sufficient resources to provide sufficient performance. Local caching on the nodes reduces the hardware requirements of the image repository, as it will reduce the amount of requests.

Network latency is another possible problem. Even if throughput is equal to the local disk, achieving similar access times in all situations is difficult. When a virtual machine sends a request for a given piece of data, the request must first be carried through the network, the data retrieved from a local disk on the image repository and the reply sent over the network. While network latency alone may be shorter than disk latency, it still creates an additional step before data can be accessed. Multi-tiered storage and caching both in the image repository and locally on the nodes of the cluster can mitigate the problem to some extent, but they add additional hardware expense and will not work in all situations. For example, if a task makes a large number of random accesses to a large pool of data, caching will be ineffective as not all of the data can be cached for fast access. The data must instead be retrieved from high storage capacity hard disks with long latencies.

As with the other disk access methods, the tasks that will run on the system and the usage pattern of the system determine whether access latency

is acceptable. The largest potential problem for this deployment method is long running tasks that randomly access large pools of data.

I/O call capture on the physical hosts

If reasonable disk I/O can be guaranteed, on-demand data transfer using a separate file system layer that redirects I/O calls from the virtual machine through the network to the image repository[3, 7, 8] is a good option. Like iSCSI snapshots, this method would allow for nearly instant deployments.

One way to implement the transfer manager in OpenNebula for this method is to create a dummy image on the physical host where the virtual machine will run. This will be done instead of copying the actual image to the host and the dummy image will be mapped to an image in the image repository. This can be done using, for example, a FUSE filesystem as was done by Nicolae *et al.*[7].

Chapter 8

Conclusions

We implemented and evaluated three transfer managers for the OpenNebula cloud middleware. One of these is based on BitTorrent and two are based on multicast. Our evaluation of these transfer managers showed that they provide significantly better performance than the default transfer managers of OpenNebula for multiple simultaneous deployments. However, the fastest method of deploying a single virtual machine is by copying the image using NFS. Due to the overhead related to bootstrapping transfers using the new methods, transferring small virtual machine images in this specific environment is also fastest using NFS.

Out of the evaluated image transfer methods, BitTorrent was the one that performed best. With a 10 GB image, it was equally fast in total deployment time as UDPcast, which was the faster of the two multicast options. However, the average per machine deployment time of BitTorrent was faster.

There is still much room for improvement in each of the implemented transfer managers. Some of the inefficiencies are due to the transfer manager model of OpenNebula not being optimal for concurrent deployments. Some are due to not fully taking into account the environment in which the deployments take place. Further development should aim to minimize the overhead caused by coordinating multiple transfers. While the current transfer managers can be optimized further, the best results would be gained by changing the transfer manager architecture completely. The transfer method should also be aware of the network topology in order to avoid bottlenecks. This would require changes to the programs used for the transfers.

We also evaluated different disk access methods for virtual machines. This evaluation was done to weigh the benefit of using smaller image formats for faster deployments against the reduced disk I/O performance that these smaller formats have. QCOW2 images, raw images and logical volumes were evaluated. As expected, the sequential read and write performance of

QCOW2 images was lower than the performance of raw images. However, random access performance was equal between the two. As using QCOW2 images can provide faster deployments, they should be used in situations where the disk I/O they provide is sufficient. One option is to use the smaller format for the operating system while placing data on a logical volume or an image file.

Bibliography

[1] P. Mell and T. Grance, "The NIST Definition of Cloud Computing," tech. rep., National Institute of Standards and Technology, Information Technology Laboratory, 2009.

[2] M. Rosenblum, "The reincarnation of virtual machines," *ACM Queue*, vol. 2, no. 5, pp. 34–40, 2004.

[3] X. Wu, Z. Shen, R. Wu, and Y. Lin, "Jump-start cloud: Efficient deployment framework for large-scale cloud applications," in *Proceedings of the 7th international conference on Distributed computing and internet technology*, ICDCIT'11, 2011.

[4] M. Hibler, L. Stoller, J. Lepreau, R. Ricci, and C. Barb, "Fast, scalable disk imaging with Frisbee," in *Proceedings of the 2003 USENIX Annual Technical Conference*, USENIX '03, 2003.

[5] M. Schmidt, N. Fallenbeck, M. Smith, and B. Freisleben, "Efficient distribution of virtual machines for cloud computing," in *Proceedings of the 18th Euromicro Conference on Parallel, Distributed, and Network-Based Processing*, PDP 2010, 2010.

[6] R. Wartel, T. Cass, B. Moreira, E. Roche, M. Guijarro, S. Goasguen, and U. Schwickerath, "Image distribution mechanisms in large scale cloud providers," in *Proceedings of the 2nd IEEE International Conference on Cloud Computing Technology and Science*, CloudCom 2010, 2010.

[7] B. Nicolae, J. Bresnahan, K. Keahey, and G. Antoniu, "Going back and forth: Efficient multideployment and multisnapshotting on clouds," in *Proceedings of the 20th international symposium on High performance distributed computing*, HPDC '11, 2011.

[8] B. Nicolae, F. Cappello, and G. Antoniu, "Optimizing multi-deployment on clouds by means of self-adaptive prefetching," in *Proceedings of the*

17th International Euro-Par Conference on Parallel Processing, Euro-Par '11, 2011.

[9] H. A. Lagar-Cavilla, J. A. Whitney, A. M. Scannell, P. Patchin, S. M. Rumble, E. de Lara, M. Brudno, and M. Satyanarayanan, "SnowFlock: Rapid virtual machine cloning for cloud computing," in *Proceedings of the 4th ACM European conference on Computer systems*, EuroSys '09, 2009.

[10] F. J. Thayer and J. D. Guttman, "Copy on write," tech. rep., The MITRE Corporation, 1995.

[11] B. Shah, "Disk performance of copy-on-write snapshot logical volumes," Master's thesis, University of British Columbia, 2006.

[12] B. Cohen, "Incentives build robustness in BitTorrent," in *Proceedings of the First Workshop on Economics of Peer-to-Peer Systems*, P2PECON '03, 2003.

[13] S. E. Deering, "Host extensions for IP multicasting." RFC 1112 (Standard), Aug. 1989. Updated by RFC 2236.

[14] J. Sundell, "rTorrent." `http://libtorrent.rakshasa.no/`, 2011. version 0.8.9.

[15] J. Sundell, "libTorrent." `http://libtorrent.rakshasa.no/`, 2011. version 0.12.9.

[16] A. Knaff, "UDPCast." `http://www.udpcast.linux.lu`, 2011. version 20110710.

[17] D. Bush, "UFTP." `http://www.tcnj.edu/~bush/uftp.html`, 2011. version 3.5.1.

[18] OpenNebula Contributors, "OpenNebula." `http://opennebula.org`, 2011. version 3.1.0.

[19] T. Rinne and T. Ylönen, "scp." `http://www.openssh.org`, 2010. version OpenSSH_5.3p1, OpenSSL 1.0.0-fips 29 Mar 2010.

[20] "OpenNebula 3.0 Guides - Managing Virtual Machines 3.0." (Online) Available: `http://opennebula.org/documentation:rel3.0:vm_guide_2`, 2011.

[21] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker, "A scalable content-addressable network," *SIGCOMM Comput. Commun. Rev.*, vol. 31, pp. 161–172, August 2001.

[22] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan, "Chord: A scalable peer-to-peer lookup service for internet applications," *SIGCOMM Comput. Commun. Rev.*, vol. 31, pp. 149–160, August 2001.

[23] A. Rowstron and P. Druschel, "Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems," in *Middleware 2001* (R. Guerraoui, ed.), vol. 2218 of *Lecture Notes in Computer Science*, pp. 329–350, Springer Berlin / Heidelberg, 2001. 10.1007/3-540-45518-3_18.

[24] B. Y. Zhao, J. D. Kubiatowicz, and A. D. Joseph, "Tapestry: An infrastructure for fault-tolerant wide-area location and routing," tech. rep., 2001.

[25] M. Izal, G. Urvoy-Keller, E. Biersack, P. Felber, A. Al Hamra, and L. Garcés-Erice, "Dissecting BitTorrent: Five months in a torrent's lifetime," in *Passive and Active Network Measurement* (C. Barakat and I. Pratt, eds.), vol. 3015 of *Lecture Notes in Computer Science*, pp. 1–11, Springer Berlin / Heidelberg, 2004. 10.1007/978-3-540-24668-8_1.

[26] B. Wei, G. Fedak, and F. Cappello, "Towards efficient data distribution on computational desktop grids with BitTorrent," *Future Generation Computer Systems*, vol. 23, no. 8, pp. 983–989, 2007.

[27] A. R. Bharambe, C. Herley, and V. N. Padmanabhan, "Analyzing and improving a BitTorrent network's performance mechanisms," in *Proceedings of the 25th IEEE International Conference on Computer Communications*, INFOCOM 2006, 2006.

[28] B. N. Levine and J. J. Garcia-Luna-Aceves, "A comparison of known classes of reliable multicast protocols," in *Proceedings of the 1996 International Conference on Network Protocols*, ICNP '96, 1996.

[29] B. Bauer, M. Fenn, and S. Goasguen, "scp-wave." `http://code.google.com/p/scp-wave/`, 2010.

[30] M. McLoughlin, "The QCOW2 image format." (Online) Available: `http://people.gnome.org/~markmc/qcow-image-format.html`.

[31] C. Tang, "FVD: a high-performance virtual machine image format for cloud," in *Proceedings of the 2011 USENIX annual technical conference*, USENIX ATC'11, 2011.

[32] L. Cherkasova and J. Lee, "FastReplica: Efficient large file distribution within content delivery networks," in *Proceedings of the 4th USENIX Symposium on Internet Technologies and Systems*, USITS'03, 2003.

[33] L. Wang, K. S. Park, R. Pang, V. Pai, and L. Peterson, "Reliability and security in the CoDeeN content distribution network," in *Proceedings of the USENIX Annual Technical Conference*, USENIX '04, 2004.

[34] K. Park and V. S. Pai, "Scale and performance in the CoBlitz large-file distribution service," in *Proceedings of the 3rd Symposium on Networked Systems Design & Implementation*, NSDI'06, 2006.

[35] K. P. Birman, M. Hayden, O. Ozkasap, Z. Xiao, M. Budiu, and Y. Minsky, "Bimodal multicast," *ACM Trans. Comput. Syst.*, vol. 17, pp. 41–88, May 1999.

[36] S. Floyd, V. Jacobson, S. McCanne, C.-G. Liu, and L. Zhang, "A reliable multicast framework for light-weight sessions and application level framing," *SIGCOMM Comput. Commun. Rev.*, vol. 25, pp. 342–356, October 1995.

[37] J. Jannotti, D. K. Gifford, K. L. Johnson, M. F. Kaashoek, and J. W. O'Toole, Jr., "Overcast: reliable multicasting with an overlay network," in *Proceedings of the 4th Symposium on Operating Systems Design and Implementation*, OSDI'00, 2000.

[38] S. K. Kasera, G. Hjálmtýsson, D. F. Towsley, and J. F. Kurose, "Scalable reliable multicast using multiple multicast channels," *IEEE/ACM Trans. Netw.*, vol. 8, pp. 294–310, June 2000.

[39] D. Kostić, A. Rodriguez, J. Albrecht, and A. Vahdat, "Bullet: High bandwidth data dissemination using an overlay mesh," *SIGOPS Oper. Syst. Rev.*, vol. 37, pp. 282–297, October 2003.

[40] V. Roca and B. Mordelet, "Design of a multicast file transfer tool on top of ALC," in *Proceedings of the Seventh International Symposium on Computers and Communications*, ISCC '02, 2002.

[41] B. Bauer, M. Fenn, and S. Goasguen, "scp-tsunami." `http://code.google.com/p/scp-tsunami/`, 2010.

[42] T. Cass, S. Goasguen, B. Moreira, E. Roche, U. Schwickerath, and R. Wartel, "CERN's virtual batch farm," in *Proceedings of the Second Cloud Computing International Conference*, CLOUDVIEWS 2010, 2010.

[43] U. Schwickerath and B. Moreira, "LxCloud infrastructure: status and lessons learned." (Online) Available: `http://indico.cern.ch/getFile.py/access?contribId=6&sessionId=6&resId=0&materialId=slides&confId=138424`, 2011.

[44] B. Claudel, G. Huard, and O. Richard, "TakTuk, adaptive deployment of remote executions," in *Proceedings of the 18th ACM international symposium on High performance distributed computing*, HPDC '09, 2009.

[45] S. Kandula, S. Sengupta, A. Greenberg, P. Patel, and R. Chaiken, "The nature of data center traffic: measurements & analysis," in *Proceedings of the 9th ACM SIGCOMM conference on Internet measurement*, IMC '09, 2009.

[46] C. E. Leiserson, "Fat-trees: universal networks for hardware-efficient supercomputing," *IEEE Trans. Comput.*, vol. 34, pp. 892–901, October 1985.

[47] R. Bindal, P. Cao, W. Chan, J. Medved, G. Suwala, T. Bates, and A. Zhang, "Improving traffic locality in BitTorrent via biased neighbor selection," in *Proceedings of the 26th IEEE International Conference on Distributed Computing Systems*, ICDCS '06, 2006.

[48] V. Aggarwal, A. Feldmann, and C. Scheideler, "Can ISPs and P2P users cooperate for improved performance?," *SIGCOMM Comput. Commun. Rev.*, vol. 37, pp. 29–40, July 2007.

[49] D. R. Choffnes and F. E. Bustamante, "Taming the torrent: a practical approach to reducing cross-ISP traffic in peer-to-peer systems," *SIGCOMM Comput. Commun. Rev.*, vol. 38, pp. 363–374, August 2008.

[50] H. Xie, Y. R. Yang, A. Krishnamurthy, Y. G. Liu, and A. Silberschatz, "P4P: provider portal for applications," *SIGCOMM Comput. Commun. Rev.*, vol. 38, pp. 351–362, August 2008.

[51] D. Engling, "opentracker." `http://erdgeist.org/arts/software/opentracker/`, 2011. latest version from version control, checked out September 29th 2011.

[52] E. Galstad, "Nagios." `http://www.nagios.org/`, 2011. version 3.3.1.

[53] A. Knaff, "[Udpcast] Reliable multicast protocol." (Online) Available: `http://www.udpcast.linux.lu/pipermail/udpcast/2005-December/000436.html`, Cited: 13.1.2012, 2005.

[54] ISO, *International standard ISO/IEC 9945: Information technology–Portable Operating System Interface (POSIX)*. 10662 Los Vaqueros Circle, Los Alamitos, California 90720-1314, USA: IEEE Computer Society Press, 2009.

[55] W. D. Norcott and D. Capps, "IOzone." `http://www.iozone.org/`, 2011. version 3.397.

[56] R. Coker and T. Bray, "bonnie++." `http://www.coker.com.au/bonnie++/`, 2008. version 1.03e.

[57] W. D. Norcott and D. Capps, "Iozone filesystem benchmark." (Online) Available: `http://www.iozone.org/docs/IOzone_msword_98.pdf`, Cited: 29.12.2011, 2011.

[58] R. Coker, "Bonnie++ documentation." (Online) Available: `http://www.coker.com.au/bonnie++/readme.html`, Cited: 29.12.2011, 2011.

[59] M. Larabel and M. Tippett, "Phoronix test suite." `http://www.phoronix-test-suite.com/`, 2011. version 3.4.0.

[60] J. Axboe, "Flexible IO Tester." `http://git.kernel.dk/?p=fio.git;a=summary`, 2011. version 1.57.

[61] D. R. Hipp, "SQLite." `http://www.sqlite.org/`, 2011. version 3.7.3.

[62] I. Gent, C. Jefferson, L. Kotthoff, I. Miguel, N. Moore, P. Nightingale, K. Petrie, and A. Rendl, "MINION." `http://minion.sourceforge.net/`, 2011. version 0.12.

[63] S. Eddy, "HMMER." `http://hmmer.janelia.org/`, 2003. version 2.3.2.

[64] P. Marciniak, N. Liogkas, A. Legout, and E. Kohler, "Small is not always beautiful," in *Proceedings of the 7th international conference on Peer-to-peer systems*, IPTPS'08, 2008.

[65] F. Bellard, "QEMU Emulator User Documentation." (Online) Available: `http://qemu.weilnetz.de/qemu-doc.html`, Cited: 30.12.2011, 2011.

[66] M. Piatek, T. Isdal, T. Anderson, A. Krishnamurthy, and A. Venkataramani, "Do incentives build robustness in BitTorrent?," in *Proceedings of the 4th USENIX Symposium on Networked Systems Design and Implementation*, 2007.

[67] T. Locher, P. Moor, S. Schmid, and R. Wattenhofer, "Free riding in BitTorrent is cheap," in *Proceedings of the Fifth Workshop on Hot Topics in Networks*, HotNets-V, 2006.

[68] J. Lee, J. Elsasser, E. Petit, and M. Livingston, "Transmission." `http://www.transmissionbt.com/`, 2011. version 2.42.

[69] "flock(1)." (Online) Available: `http://linux.die.net/man/1/flock`, Cited: 5.1.2012, 2011.

[70] "util-linux." (Online) Available: `https://github.com/karelzak/util-linux`, Cited: 5.1.2012, 2011.

[71] "flock(2)." (Online) Available: `http://www.kernel.org/doc/man-pages/online/pages/man2/flock.2.html`, Cited: 5.1.2012, 2011.

[72] "Btrfs." (Online) Available: `https://btrfs.wiki.kernel.org/`, Cited: 9.1.2012, 2011.

[73] J. Bonwick and B. Moore, "ZFS: The last word in file systems." (Online) Available: `http://hub.opensolaris.org/bin/download/Community+Group+zfs/docs/zfslast.pdf`, Cited: 9.1.2012, 2007.

[74] B. Behlendorf, "ZFS on Linux." (Online) Available: `http://zfsonlinux.org/`, Cited: 9.1.2012, 2011.

[75] J. Satran, K. Meth, C. Sapuntzakis, M. Chadalapaka, and E. Zeidner, "Internet Small Computer Systems Interface (iSCSI)." RFC 3720 (Proposed Standard), Apr. 2004. Updated by RFCs 3980, 4850, 5048.

Appendix A

Transfer manager source codes

The SSH transfer manager that is included with OpenNebula was taken as the basis for the source codes of the new implementations. Most of the scripts were used as is, as they are not used in the deployment process that we discuss in this thesis. The script that contains the logic described in Chapter 4 of this thesis is called *tm_clone.sh*. The contents of this script for each of the new transfer managers is included here. While most of the code was replaced with a customized implementation, some of the code in the beginning of each of the listings is taken as is from the SSH transfer manager (about 20 lines). It is included here for completeness, context and to provide a version of each of the scripts that works without modification. These scripts have been tested to work in OpenNebula 3.0 and should be usable as listed here excluding some trivial changes to the other scripts and settings files.

This notice was omitted from each of the listings to avoid repetition and is included here to comply with the Apache License:

```
# ------------------------------------------------------------------------- #
# Copyright 2002-2011, OpenNebula Project Leads (OpenNebula.org)            #
#                                                                           #
# Licensed under the Apache License, Version 2.0 (the "License"); you may   #
# not use this file except in compliance with the License. You may obtain   #
# a copy of the License at                                                  #
#                                                                           #
# http://www.apache.org/licenses/LICENSE-2.0                                #
#                                                                           #
# Unless required by applicable law or agreed to in writing, software       #
# distributed under the License is distributed on an "AS IS" BASIS,         #
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.  #
# See the License for the specific language governing permissions and       #
# limitations under the License.                                            #
#-------------------------------------------------------------------------- #
```

A.1 BitTorrent

```
#!/bin/bash

SRC=$1
DST=$2

if [ -z "${ONE_LOCATION}" ]; then
    TMCOMMON=/usr/lib/one/mads/tm_common.sh
else
    TMCOMMON=$ONE_LOCATION/lib/mads/tm_common.sh
fi

. $TMCOMMON

get_vmdir

SRC_PATH=`arg_path $SRC`
DST_PATH=`arg_path $DST`

SRC_HOST=`arg_host $SRC`
DST_HOST=`arg_host $DST`

SRC_BASE=`basename $SRC`
DST_BASE=`basename $DST`

SRC_TORRENT=$SRC_BASE.torrent

# Override for testing
LOCALCOPY="/bin/cp"

log_debug "$1 $2"
log_debug "DST: $DST_PATH"

DST_DIR=`dirname $DST_PATH`

log "Creating directory $DST_DIR"
exec_and_log "$SSH $DST_HOST mkdir -p $DST_DIR" \
    "Error creating directory $DST_DIR"

case $SRC in
http://*)
    log "Downloading $SRC"
    exec_and_log "$SSH $DST_HOST $WGET -O $DST_PATH $SRC" \
        "Error downloading $SRC"
    ;;

*)
    log "Checking for existing image"
    $SSH $DST_HOST ls $VMDIR/$SRC_BASE.finished &> /dev/null
    if [ "$?" -ne "0" ]; then
        log "No existing image found, going to transfer"
    else
        $SSH $DST_HOST ls $VMDIR/$SRC_BASE &> /dev/null
        if [ "$?" -ne "0" ]; then
            log ".finished notification file found but no image found"
            log "Going to delete .finished file and proceed as usual"
            $SSH $DST_HOST rm -f $VMDIR/$SRC_BASE.finished
        else
            log "Found existing image, going to copy"
```

```
            exec_and_log "$SSH $DST_HOST $LOCALCOPY $VMDIR/$SRC_BASE
                $DST_PATH" \
                "Error copying $VMDIR/$SRC_BASE"
61          exec_and_log "$SSH $DST_HOST chmod g+rw $VMDIR/$SRC_BASE*"
            exec_and_log "$SSH $DST_HOST chmod a+rw $DST_PATH"
            log_debug "`date +%H:%M:%S.%N`, clone finished"
            exit 0
        fi
66  fi

    log "Cloning $SRC"
    exec_and_log "$SCP $ONE_LOCATION/torrents/$SRC_TORRENT
        $DST_HOST:/usr/etc/rtorrent/torrents" \
        "Error copying $SRC to $DST"
71
    log "Waiting for torrent to finish"
    $SSH $DST_HOST ls $VMDIR/$SRC_BASE.finished &> /dev/null
    while [ "$?" -ne "0" ]
    do
76              sleep 5
                $SSH $DST_HOST "touch
                    /usr/etc/rtorrent/torrents/$SRC_TORRENT; \
                                                ls $VMDIR/$SRC_BASE.finished
                                                    &> /dev/null"
    done
    log "Got notification from torrent client that the download is finished"
81
    exec_and_log "$SSH $DST_HOST $LOCALCOPY $VMDIR/$SRC_BASE $DST_PATH" \
        "Error copying $VMDIR/$SRC_BASE"

        exec_and_log "$SSH $DST_HOST chmod g+rw $VMDIR/$SRC_BASE*"
86  ;;
esac

exec_and_log "$SSH $DST_HOST chmod a+rw $DST_PATH"
log_debug "`date +%H:%M:%S.%N`, clone finished"
```

A.2 UDPcast

```
#!/bin/bash

set -u
set -e

SRC=$1
DST=$2

if [ -z "${ONE_LOCATION}" ]; then
    TMCOMMON=/usr/lib/one/mads/tm_common.sh
else
    TMCOMMON=$ONE_LOCATION/lib/mads/tm_common.sh
fi

. $TMCOMMON

get_vmdir

SRC_PATH=`arg_path $SRC`
DST_PATH=`arg_path $DST`

SRC_HOST=`arg_host $SRC`
DST_HOST=`arg_host $DST`

SRC_BASE=`basename $SRC`
DST_BASE=`basename $DST`

VMID=`echo $DST_PATH | cut -d"/" -f5`
DU="du --apparent-size -m"
IMG_SIZE=`$DU $SRC_PATH | $AWK '{print \$1}'`

# This file contains the list of image destinations for a given host.
# Each deployment script adds its own destination to this, and it is then
# read by the instance of the deployment script that launches the receiver
# on a host.
DSTLIST_FILE=/tmp/one-udpcast-dstlist-$DST_HOST.txt

# Operations on the destination list file must be locked to prevent
# corrupted writes and to prevent late arrivals from thinking their file
# will actually arrive at its destination.
DSTLIST_FILE_LOCK=/tmp/one-udpcast-dstlist-$DST_HOST-lockdir

# Only one receiver per host should be launched. This lock is acquired so
# that only one instance at a time gets to launch a receiver on a given
# host.
HOST_RECEIVER_LOCK=/tmp/one-udpcast-rcvr-$DST_HOST-lockdir

# The lock for the instance that launches the sender.
TRANSFER_LOCK=/tmp/one-udpcast-transfer-lockdir-$SRC_BASE

# A global lock to prevent multiple instances of Udpcast.
UDPCAST_LOCK=/tmp/one-udpcast-lockdir

# Some constants to make the code easier to read.
# These control whether a lock is waited for or not.
WAIT="1"
DO_NOT_WAIT="0"
```

```
# Override for testing
#LOCALCOPY="/bin/cp"

# These three lines can be used to redirect the whole output of this
# script to the listed location. This is useful for debugging.
CURRENT_DATE=`date +%F-%k:%M.%S`
LOG_LOCATION=/srv/cloud/one/transferlogs/$CURRENT_DATE-$$.log
#exec > $LOG_LOCATION 2>&1

# Removes the destination list file from both the frontend and
# the destination host.
function remove_dst_file() {
    rm -f $DSTLIST_FILE
    $SSH $DST_HOST rm -f $DSTLIST_FILE
}

# General locking function with stale lock detection (using PID files)
# and two modes of operation: one where the caller will wait for the lock
# and another where it simply checks if the lock has been acquired by
# someone else.
function acquire_lock() {
    PIDFILE=$1/PID
    WILLWAIT=$2
    if ! mkdir $1 &> /dev/null; then
        if ! LOCKERPID=`cat ${PIDFILE}`; then
            # If the PID file does not exist, it probably means
            # that the lock was just acquired, was just released or
            # was just reacquired. In any case, it is best to wait.
            log "Lockdir exists but pidfile does not, going to wait for
                lock"
            case $WILLWAIT in
                0)
                    return
                    ;;
                1)
                    wait_for_lock $1
                    ;;
            esac
        else
            if ! kill -0 $LOCKERPID &>/dev/null; then
                # The lock was stale, remove the old PID file
                # and assume its safe to proceed
                log "Stale lockdir found, going to take the lock"
                rm -f $PIDFILE
                echo "$$" >> ${PIDFILE}
            else
                case $WILLWAIT in
                    0)
                        return
                        ;;
                    1)
                        wait_for_lock $1
                        ;;
                esac
            fi
        fi
    else
        echo "$$" >> ${PIDFILE}
    fi
}

# Periodically check for the availability of a lock and acquire it
```

```
# as soon as it becomes available.
# Note: The absence of a FIFO queue here might lead to starvation, so
# it might not be a good idea to use this in some general purpose locking
# utility. It SHOULD work in the context of the transfer manager.
function wait_for_lock() {
    GOT_LOCK=0
    PIDFILE=$1/PID
    while [ "$GOT_LOCK" -eq 0 ]
    do
        if mkdir $1 &> /dev/null; then
            echo "$$" >> ${PIDFILE}
            log "`date +%H:%M:%S.%N`, got lock after wait: $1"
            GOT_LOCK=1
        else
            sleep 1
        fi
    done
}

# Releases a lock by removing the corresponding lock directory
function release_lock() {
    rm -rf $1
    log "`date +%H:%M:%S.%N`, released lock: $1"
}

# Halt until the lock given as the first parameter has been released
function wait_on_lock() {
    while [ -d $1 ]; do
        sleep 5
    done
}

# Wait for an incomplete image to become complete or timeout
# if it does not do so in a timely manner
function wait_for_complete_image {
    ACT_SIZE=`$SSH $DST_HOST $DU $VMDIR/$SRC_BASE | $AWK '{print \$1}'`
    COUNTER=0
    while [ "$ACT_SIZE" -ne "$IMG_SIZE" ]; do
        sleep 1
        ACT_SIZE=`$SSH $DST_HOST $DU $VMDIR/$SRC_BASE | $AWK '{print \$1}'`
        let COUNTER=COUNTER+1
        if [ "$COUNTER" -gt "40"  ]; then
            log "Timeout while waiting for the image to be completed"
            return 1
        fi
    done
    return 0
}

# Check for an existing image at the root of the VM directory of a node
function check_existing_image {
    $SSH $DST_HOST ls $VMDIR/$SRC_BASE &> /dev/null
    if [ "$?" -eq "0" ]; then
        log "Image $SRC_BASE is already on the host"
        ACT_SIZE=`$SSH $DST_HOST $DU $VMDIR/$SRC_BASE | $AWK '{print \$1}'`
        if [ "$ACT_SIZE" -ne "$IMG_SIZE" ]; then
            wait_for_complete_image
        fi
        if [ "$?" -eq "0" ]; then
            log "Making a bootable copy of image"
            exec_and_log "$SSH $DST_HOST $LOCALCOPY $VMDIR/$SRC_BASE
                $DST_PATH" \
```

```
                "Local copy of image from $VMDIR/$SRC_BASE to $DST failed"
            log "`date +%H:%M:%S.%N`, clone finished"
            exit 0
        fi
    fi
}

function start_remote_receiver {
        $SSH $DST_HOST "nohup $UDP_RECEIVER --nokbd \
        --pipe \"tee `$SSH $DST_HOST cat $DSTLIST_FILE | xargs`\" \
        --interface $RECEIVER_INTERFACE &> /dev/null &"
}

log "$1 $2"
log "DST: $DST_PATH"

DST_DIR=`dirname $DST_PATH`

log "Creating directory $DST_DIR"
exec_and_log "$SSH $DST_HOST mkdir -p $DST_DIR" \
    "Error creating directory $DST_DIR"

case $SRC in
http://*)
    log "Downloading $SRC"
    exec_and_log "$SSH $DST_HOST $WGET -O $DST_PATH $SRC" \
        "Error downloading $SRC"
    ;;

*)
    DST_ADDED=0
    while [ "$DST_ADDED" -eq 0 ]; do
        if acquire_lock ${DSTLIST_FILE_LOCK} $DO_NOT_WAIT; then
            trap "release_lock ${DSTLIST_FILE_LOCK}; exit" INT TERM EXIT
            log "adding destination filename: `date +%H:%M:%S.%N`"
            echo $DST_PATH >> $DSTLIST_FILE
            DST_ADDED=1
            log "Added destination to host's destination list"
            release_lock ${DSTLIST_FILE_LOCK}
            trap - INT TERM EXIT
        else
            sleep 0.7
        fi
    done

    log "Trying to acquire transfer lock"
    if acquire_lock ${TRANSFER_LOCK} $DO_NOT_WAIT; then
        trap "release_lock ${TRANSFER_LOCK}; exit" INT TERM EXIT
        acquire_lock ${HOST_RECEIVER_LOCK} $WAIT
        trap - INT TERM EXIT
        trap "release_lock ${TRANSFER_LOCK}; \
              release_lock ${HOST_RECEIVER_LOCK}; \
              exit" INT TERM EXIT
        log "`date +%H:%M:%S.%N`, Got transfer lock, going to transfer"
        sleep $TRANSFER_START_WAIT
        acquire_lock ${DSTLIST_FILE_LOCK} $WAIT
        trap - INT TERM EXIT
        trap "release_lock ${TRANSFER_LOCK}; \
              release_lock ${HOST_RECEIVER_LOCK}; \
              release_lock ${DSTLIST_FILE_LOCK}; \
              exit" INT TERM EXIT
        log "Locked list of destinations"
```

```
        acquire_lock ${UDPCAST_LOCK} $WAIT
        trap - INT TERM EXIT
        trap "release_lock ${TRANSFER_LOCK}; \
              release_lock ${HOST_RECEIVER_LOCK}; \
              release_lock ${DSTLIST_FILE_LOCK}; \
              release_lock ${UDPCAST_LOCK}; \
              remove_dst_file; \
              exit" INT TERM EXIT
        log "Got Udpcast lock, cloning"
        $SCP $DSTLIST_FILE $DST_HOST:$DSTLIST_FILE
        start_remote_receiver
        $UDP_SENDER --nokbd --full-duplex \
            --interface $SENDER_INTERFACE \
            --autostart $AUTOSTART_TIME \
            --slice-size $SLICE_SIZE \
            -f $SRC_PATH &> $ONE_LOCATION/transferlogs/$VMID-sender.log
        remove_dst_file
        release_lock ${UDPCAST_LOCK}
        release_lock ${DSTLIST_FILE_LOCK}
        release_lock ${HOST_RECEIVER_LOCK}
        release_lock ${TRANSFER_LOCK}
        trap - INT TERM EXIT
    else
        log "Did not get transfer lock, trying to get host receiver lock"
        if acquire_lock ${HOST_RECEIVER_LOCK} $DO_NOT_WAIT; then
            trap "release_lock ${HOST_RECEIVER_LOCK}; \
                  remove_dst_file; exit" INT TERM EXIT
            log "Got host receiver lock, going to start receiver on host"
            sleep $TRANSFER_START_WAIT
            acquire_lock ${DSTLIST_FILE_LOCK} $WAIT
            trap - INT TERM EXIT
            trap "release_lock ${HOST_RECEIVER_LOCK}; \
                  release_lock ${DSTLIST_FILE_LOCK}; \
                  remove_dst_file; \
                  exit" INT TERM EXIT
            log "Locked list of destinations"
            $SCP $DSTLIST_FILE $DST_HOST:$DSTLIST_FILE
            start_remote_receiver
            remove_dst_file
            wait_on_lock ${TRANSFER_LOCK}
            release_lock ${DSTLIST_FILE_LOCK}
            release_lock ${HOST_RECEIVER_LOCK}
            trap - INT TERM EXIT
        else
            log "Did not get host receiver lock, going to wait for transfer
                to finish"
            wait_on_lock ${TRANSFER_LOCK}
        fi
    fi
esac

exec_and_log "$SSH $DST_HOST chmod a+rw $DST_PATH" \
        "Unable to change permissions for $DST_PATH"
log "`date +%H:%M:%S.%N`, clone finished"
```

A.3 UFTP

```
#!/bin/bash

SRC=$1
DST=$2

if [ -z "${ONE_LOCATION}" ]; then
    TMCOMMON=/usr/lib/one/mads/tm_common.sh
else
    TMCOMMON=$ONE_LOCATION/lib/mads/tm_common.sh
fi

. $TMCOMMON

get_vmdir

SRC_PATH=`arg_path $SRC`
DST_PATH=`arg_path $DST`

SRC_HOST=`arg_host $SRC`
DST_HOST=`arg_host $DST`

SRC_BASE=`basename $SRC`
DST_BASE=`basename $DST`

TEMP_DEST=$VMDIR/$SRC_BASE

VMID=`echo $DST_PATH | cut -d"/" -f5`
DU="du --apparent-size -m"
IMG_SIZE=`$DU $SRC_PATH | $AWK '{print \$1}'`

UFTP_LOCK=/tmp/one-uftp-lockdir
TRANSFER_LOCK=/tmp/one-transfer-lockdir-$SRC_BASE
HOST_FILE_LOCK=/tmp/one-hostfile-lockdir-$SRC_BASE
HOSTS_FILE=/tmp/$SRC_BASE-hosts.txt
PID_FILE=/tmp/$SRC_BASE-transfer.pid
TRANSFER_EXIT_STATUS_FILE=/tmp/$SRC_BASE-one-transfer-exit-status

# Some constants to make the code easier to read.
# These control whether a lock is waiter for or not.
WAIT="1"
DO_NOT_WAIT="0"

# Override for testing
#LOCALCOPY="/bin/cp"

TRANSFER_START_WAIT=30

# Parameters for UFTP
TRANSFER_SPEED=800000
MAX_TRANSFER_TIME=10000
INTERFACE=eth1
FRAME_SIZE=1472
TTL=2
MC_GROUP=230.5.5.x
ANNOUNCE_TIME=15
STATUS_TIME=30
ANNOUNCE_INTERVAL=700
STATUS_INTERVAL=700
```

```
REGISTER_INTERVAL=1200
DONE_INTERVAL=1200
BUFFER_SIZE=104857600
LOG_LEVEL=5
LOG_LOCATION=/tmp/uftp.log

CURRENT_DATE=`date +%F-%k:%M.%S`
LOG_LOCATION=/srv/cloud-devel/one/transferlogs/$CURRENT_DATE-$$.log

#exec > $LOG_LOCATION 2>&1

function acquire_lock() {
    PIDFILE=$1/PID
    WILLWAIT=$2
    if ! mkdir $1 &> /dev/null; then
        if ! LOCKERPID=`cat ${PIDFILE}`; then
            # If the PID file does not exist, it probably means
            # that the lock was just acquired, was just released or
            # was just reacquired. In any case, it is best to wait.
            log_debug "Lockdir exists but pidfile does not, going to wait
                for lock"
            case $WILLWAIT in
                0)
                    return
                    ;;
                1)
                    wait_for_lock $1
                    ;;
            esac
        else
            if ! kill -0 $LOCKERPID &>/dev/null; then
                # The lock was stale, remove the old PID file
                # and assume its safe to proceed
                log_debug "Stale lockdir found, going to take the lock"
                rm -f $PIDFILE
                echo "$$" >> ${PIDFILE}
            else
                case $WILLWAIT in
                    0)
                        return
                        ;;
                    1)
                        wait_for_lock $1
                        ;;
                esac
            fi
        fi
    else
        echo "$$" >> ${PIDFILE}
    fi
}

function wait_for_lock() {
    GOT_LOCK=0
    PIDFILE=$1/PID
    while [ "$GOT_LOCK" -eq 0 ]
    do
        if mkdir $1 &> /dev/null; then
            echo "$$" >> ${PIDFILE}
            log_debug "`date +%H:%M:%S.%N`, got lock after wait: $1"
            GOT_LOCK=1
        else
```

```
            sleep 1
        fi
    done
}

function release_lock() {
    rm -rf $1
    log_debug "`date +%H:%M:%S.%N`, released lock: $1"
}

function wait_on_lock() {
    while [ -d $1 ]; do
        sleep 5
    done
}

function wait_for_complete_image {
    ACT_SIZE=`$SSH $DST_HOST $DU $VMDIR/$SRC_BASE | $AWK '{print \$1}'`
    COUNTER=0
    while [ "$ACT_SIZE" -ne "$IMG_SIZE" ]; do
        sleep 1
        ACT_SIZE=`$SSH $DST_HOST $DU $VMDIR/$SRC_BASE | $AWK '{print \$1}'`
        let COUNTER=COUNTER+1
        if [ "$COUNTER" -gt "40"  ]; then
            log "Timeout while waiting for the image to be completed"
            return 1
        fi
    done
    return 0
}

function check_existing_image {
    $SSH $DST_HOST ls $VMDIR/$SRC_BASE &> /dev/null
    if [ "$?" -eq "0" ]; then
        log "Image $SRC_BASE is already on the host"
        ACT_SIZE=`$SSH $DST_HOST $DU $VMDIR/$SRC_BASE | $AWK '{print \$1}'`
        if [ "$ACT_SIZE" -ne "$IMG_SIZE" ]; then
            wait_for_complete_image
        fi
        if [ "$?" -eq "0" ]; then
            log "Making a bootable copy of image"
            exec_and_log "$SSH $DST_HOST $LOCALCOPY $VMDIR/$SRC_BASE
                $DST_PATH" \
                "Local copy of image from $VMDIR/$SRC_BASE to $DST failed"
            log "`date +%H:%M:%S.%N`, clone finished"
            exit 0
        fi
    fi
}

log_debug "$1 $2"
log_debug "DST: $DST_PATH"

DST_DIR=`dirname $DST_PATH`

log "Creating directory $DST_DIR"
exec_and_log "$SSH $DST_HOST mkdir -p $DST_DIR" \
    "Error creating directory $DST_DIR"

case $SRC in
http://*)
    log "Downloading $SRC"
```

```
    exec_and_log "$SSH $DST_HOST $WGET -O $DST_PATH $SRC" \
        "Error downloading $SRC"
    ;;

*)
    # First a check to see if the image is already there
    check_existing_image

    set -e
    HOSTNAME_ADDED=0
    while [ "$HOSTNAME_ADDED" -eq 0 ]; do
        if acquire_lock ${HOST_FILE_LOCK} $DO_NOT_WAIT; then
            trap "release_lock ${HOST_FILE_LOCK}; exit" INT TERM EXIT
            log_debug "adding hostname: `date +%H:%M:%S.%N`"
            echo $DST_HOST >> $HOSTS_FILE
            HOSTNAME_ADDED=1
            log "Added hostname to transfer list"
            release_lock ${HOST_FILE_LOCK}
            trap - INT TERM EXIT
        else
            sleep 0.1
        fi
    done

    # Another check, if a transfer has finished in the meanwhile
    set +e
    check_existing_image
    set -e

    log "Trying to acquire transfer lock"
    if acquire_lock ${TRANSFER_LOCK} $DO_NOT_WAIT; then
        trap "release_lock ${TRANSFER_LOCK}; exit" INT TERM EXIT
        log "`date +%H:%M:%S.%N`, Got transfer lock, going to transfer"
        sleep $TRANSFER_START_WAIT
        acquire_lock ${HOST_FILE_LOCK} $WAIT
        trap - INT TERM EXIT
        trap "release_lock ${TRANSFER_LOCK}; \
              release_lock ${HOST_FILE_LOCK}; \
              rm -f ${HOSTS_FILE}; \
              exit" INT TERM EXIT
        log "Locked hosts file"
        log "Cloning $SRC"

        acquire_lock ${UFTP_LOCK} $WAIT
        trap - INT TERM EXIT
        trap "release_lock ${TRANSFER_LOCK}; \
              release_lock ${HOST_FILE_LOCK}; \
              release_lock ${UFTP_LOCK}; \
              rm -f ${HOSTS_FILE}; \
              exit" INT TERM EXIT
        set +e
        $UFTP -R $TRANSFER_SPEED \
            -B $BUFFER_SIZE \
            -I $INTERFACE \
            -b $FRAME_SIZE \
            -t $TTL \
            -P $MC_GROUP \
            -A $ANNOUNCE_TIME \
            -S $STATUS_TIME \
            -a $ANNOUNCE_INTERVAL \
            -s $STATUS_INTERVAL \
            -r $REGISTER_INTERVAL \
```

```
            -d $DONE_INTERVAL \
            -W $MAX_TRANSFER_TIME \
            -H @$HOSTS_FILE \
            $SRC_PATH &> $ONE_LOCATION/transferlogs/$VMID.log

        echo $? > $TRANSFER_EXIT_STATUS_FILE
        set -e
        release_lock ${UFTP_LOCK}
        rm -f $HOSTS_FILE
        log_debug "`date +%H:%M:%S.%N`, removed hosts.txt"
        release_lock ${TRANSFER_LOCK}
        release_lock ${HOST_FILE_LOCK}
        trap - INT TERM EXIT
    else
        log "Did not get lock, going to wait"
        wait_on_lock ${TRANSFER_LOCK}
    fi

    TRANSFER_EXIT_STATUS=`cat $TRANSFER_EXIT_STATUS_FILE`
    if [ "$TRANSFER_EXIT_STATUS" -eq "0" ]; then
        log "Transfer was succesful, going to make a bootable copy of the
            image"
        exec_and_log "$SSH $DST_HOST $LOCALCOPY $VMDIR/$SRC_BASE $DST_PATH"
            \
            "Local copy of image from $VMDIR/$SRC_BASE to $DST failed"
    else
        log "Transfer failed"
        exit 2
    fi
esac

exec_and_log "$SSH $DST_HOST chmod a+rw $DST_PATH" \
        "Unable to change permissions for $DST_PATH"
exec_and_log "$SSH $DST_HOST chmod g+rw $VMDIR/$SRC_BASE" \
        "Unable to change permissions for $VMDIR/$SRC_BASE"
log_debug "`date +%H:%M:%S.%N`, clone finished"
```