Teleporting virtual machines

Master System and Network Engineering

Authors:
Harm Dermois
`harm.dermois@os3.nl`
Carlo Rengo
`carlo.rengo@os3.nl`

Supervisors:
Oskar van Deventer (TNO)
`oskar.vandeventer@tno.nl`
Jan Sipke van der Veen (TNO)
`jan_sipke.vanderveen@tno.nl`

Universiteit van Amsterdam

February 8, 2015

Abstract

The research described in this paper is about Virtual Machine (VM) teleportation, a way of copying Virtual Machines across two nodes that tries to minimize the bandwidth usage between them. This “teleportation” technique heavily relies on using “similar” sources of data, such as local copies of Virtual Machines on the destination node or software repositories available on the Internet, to recreate the original Virtual Machine.

From the research done and the Proof of Concept written during a one-month project, it can be concluded that with VM teleportation bandwidth is indeed saved and, in particular cases, teleportation may even be faster than a normal copy.

Acknowledgements

We would like to thank Oskar van Deventer and Jan Sipke van der Veen for helping us with our project. Their assistance and guidance during the whole research period have been really valuable to us.

We would also like to thank TNO for giving us the opportunity to do this research at their office in Delft and lending us the materials needed to perform the research.

Contents

1. Introduction

Virtual machines (VM) are a key unit of cloud infrastructures. Often, VMs need to be moved or replicated between hosts, in a way similar to what happens in a Content Delivery Network (CDN).

However, VMs are harder to distribute with current CDN technologies, which are developed with static content in mind. Virtual machine disks are binaries of considerable size and a small change in the VM filesystem will make a completely different binary, thus making classic caching techniques not really useful. As nodes can be located anywhere in the world, the transport of a whole Virtual Machine can take significant amounts of time and data-transport capacity.

A hypothesis is that time and transport efficiency could be gained by transporting only the user data (databases, files, etc...) and Operating System (OS) configuration parameters, and rebuilding the VM from a locally available copy of the OS and installed applications.

In this project, this hypothesis will be investigated by implementing a VM "teleporter" as described above, as well as a system that transports a VM in bulk. The teleported VM will be a "functional" copy 4.1 of the source: it should work, look and behave the same, but it doesn't have to be an exact replica (for instance, installed packages don't have to be the same versions of the source ones).

1.1. Research question

- Is it technically viable to implement a VM teleporting system that can be convenient to setup in a real world scenario?
- Is the required data for transferring a virtual machine indeed less for a teleported one than for a conventionally migrated one?
- Is a teleported VM indeed quicker up-and-running than a conventionally migrated one?

Essential to the research question are a number of subquestions:

- When is a virtual machine considered to be a "functional copy" of another virtual machine?
- How can data on a (virtual) disk be logically divided into user data and Operating System data?
- Which best practices should be followed, in a virtual environment, to minimize bandwidth consumption and speed-up the teleportation?

1.2. Context

VM teleportation only copies the data that is needed and lets the destination get parts of the VM itself. This will reduce the amount of data that is being send from the source to the destination. This might be desirable in situation where bandwidth is an issue. Bandwidth issues can occur when servers are geographically far apart, or in regions having poor network performance. In this situation sending as little data as possible is preferred. VM teleportation might help in these cases by letting the destination server get software packages itself. These packages are in most cases local and quick to retrieve.

Another use case might be a Virtual Machine Delivery Network (VMDN) which is explained in detail in reference [1].

In general, use cases for VM teleportation involve having limited bandwidth between the source and the destination nodes. Teleportation can also be used to speed up the delivery of a VMs. These are situations where there are already other local VMs available. These cases can occur when there is a need for multiple applications which need to be separated from each other even if they run on the same Operating System (OS). This can also be used when you want to have more capacity for one application and run multiple VMs with the same OS.

1.3. Scope

A Proof of Concept (PoC) will be written to “smartly” migrate a VM from one hypervisor to another. Some limitations will be put as fringe cases, like external repositories and retro-compatibility issues, will not be considered. The focus will be less on “unorthodox” manual software installations and it is not the aim of the project to make a completely generic solution. For the sake of simplicity, only common Linux distributions (Ubuntu and CentOS) and Kernel-based Virtual Machine (KVM)[2] hypervisor will be taken into account. Ubuntu and CentOS were chosen, because these distros are very popular and we have experience using them. KVM was chosen because it requires no special configuration to run and it is required for libguestfs (see 4.3). Despite the simplicity desired, the aim of this project is to keep it as generic as possible, this is why the decision of not writing on the teleported VM virtual disk has been made; for this reason also some “handy” tools were deliberately avoided (see 3.2).

It is also worth to mention that only cold (powered off) VMs with default virtual hardware and no more than one virtual disk are used as candidates for the VM teleportation: as only one month was given to do this research project, these decisions were made to prevent potential technical problems, simplifying the setup of the test infrastructure and quicken the writing of the PoC we use later in this project.

2. Previous work

This work is based on a Master thesis[1] which has been previously written as a TNO project. This paper introduces Virtual Machine Delivery Networks (VMDNs) and proposes three different strategies of implementing a VMDN.

The first strategy works with data clusters [3] while the second works with files and cached files. The last strategy, which will be used in this project, is based on the concept of “functional” copy 4.1.

Another interesting paper is [4], the result of which shows the possibility of live migrating VMs over long geographical distances with a downtime comparable to intra-LAN local migrations (about 5-10 times greater despite 1000 times higher round-trip times. The research discussed in this paper, however, tries a different approach based on the concept of “functional copy”4.1 and by testing a VM migration over ordinary WAN channels.

3. Background

This section contains basic information for the topic discussed in this paper.

3.1. Migration techniques

Common hypervisors such as Xen, KVM and VMware can do live migration (that is, without even powering off) of a VM, but the main requirement is to have the virtual disk image file(s) locally accessible from both the nodes. More details are available in [5] and [6].

For any other eventuality, different methods exist but, in short, they all do a normal copy of the VM disk image file(s), plus other metadata (see [7]). For instance, KVM tools can dump Virtual Machine information using the command `virsh dumpxml` ([8]). Another tool worth to be mentioned is VMware vCenter Converter [9], which can also convert different types of virtual disk.

3.2. Configuration management and VM provisioning

Two of the most crucial aspect in VM teleportation are describing and building a VM. While the former heavily relies on the concept of functional copy that is being described subsequently in 4.1, the latter (also known as "provisioning", a term that also involves customizing) is something easily accessible with the current technologies and, mostly, is just a matter of wisely using already available tools. Tools worth to be mentioned are Puppet[10] and Vagrant[11].

Puppet is a configuration management system that allows to define the state of an IT infrastructure, then automatically enforces the correct state. The configuration of any node of the infrastructure is kept in Puppet specific files which give information about what software should be installed and the configuration of that software. These Puppet files can also be used to configure a machine from scratch.
The downside of using this tool in the project is that each VM should have Puppet installed along with the necessary configuration files. Not only does this require you have to remove the installation afterwards. It also makes Puppet mandatory to be ran on each VM. It will also make the VM teleportation software dependent on Puppet. This undesirable in a market that advances this quickly. Although Puppet have some utilities to show what packages are installed (useful for the description, see 4.1), this is no more than what common Linux OS package managers can show. One of the aims of this project is to make the teleportation of a VM as generic as possible.

Vagrant is a tool used for VM provisioning. It is primarily used to quickly create development environments. This is done thanks to a vagrant file which describes the provisioning of the VM. Vagrant can work in conjunction with Puppet to build and configure a Virtual Machine from a description. After some analysis of the tool, the decision has been made to use more generic libraries, but the approach will be very similar. Using more generic tools will give the teleportation software more control over the VM. The tools/libraries chosen are described in 4.3.

4. Methodology

As specified in 1.3, the source VM should not be changed. With this decision in mind the following software tools have been chosen (see 4.3). This section gives an introduction in the software tools used. These are tools that are generic and give some flexibility in the choice of hypervisor to manage the VMs.

Because this project is a proof of concept, Python[12] and Bash[13] language have been chosen to write the smart migrate software 5. This is a good combination for this project, as Python is a good language for fast prototyping and, since most tools used are command-line tools which makes bash scripting very handy and useful to automate the process.

4.1. Functional copy

In the introduction it is mentioned that the teleported VM should be a "functional copy" of the original. Although not a strictly defined concept, it is safe to say that a functional copy should "feel", behave and work the same as the original.

It is hard to figure out if two VMs are functionally the same. There are many factors which can change the way the VM behaves, like the version of the packages and their configuration, some temporary files, even the filesystem support of certain features. In the Proof of Concept a description is made with minimal set of information needed to recreate the VM. This description file only shows which packages are installed and not the exact version. Just having the name of the package should not give any problems in most cases, as the software repositories of well known distributions always take backwards compatibility into account (their updates are just patches and bugfixing). It would have been impossible to make the new VM install the exact version of the source packages, because the repositories only maintain the last versions of them. All that can be done other than just testing everything is trust that the distributions do their work properly.

As described in 1.3, problems related with third party repositories will not be considered. The checks done is to see if main services of the VM are running properly (e.g. does the website still look the same? is the database running and filled with the same data?) were done manually. It would have taken too much time to automate this controls, and it would have been out of the scope.

Ultimately, besides installing the same packages, our PoC copies all the data except of the package contents. This "formula" of having the same user data and configuration but possibly different minor version of system packages is our non scientific definition of functional copy.

4.2. Physical tools and setup

Unfortunately, there wasn't the possibility to test our PoC on production hypervisors (with ideally various Virtual Machines having different Operating Systems and snapshots). Due to time constrains, the measurements shown in 6 are based on two different

VMs, built with the following characteristics:

OS	CentOS Linux
Version	7.1
Architecture	64bit
Software installed	Apache - Web server ISPconfig - Hosting panel
Network configuration	DHCP
User data (1)	2 websites with database
User data (2)	2 websites with database, 9GiB random files
Disk usage (1)	2 out of 30GiB
Disk usage (2)	11 out of 30GiB

Table 1: Virtual Machines description

ISPConfig [14] was chosen as the main software on the VMs because it has dependencies, both from the CentOS repository, and from other software/libraries compiled by hand, so it could be a good "benchmark" for our tests. The only difference in the second Virtual Machine is the presence of 9GiB of extra data, that was randomly generated (see C for more details). While creating the VM, snapshots were taken after each big change made. This left us with the base image (minimal OS installation with updates) and three snapshots.

These are the two hypervisors, the source and the destination of the teleportation:

Model	Dell System XPS L702X
CPU	Intel®Core™i7-2620M CPU @ 2.70GHz (Dual Core)
Memory	8GiB RAM SODIMM DDR3 Synchronous 1333
Disk	Seagate ST9500420AS - 500GB (non SSD)
OS	Ubuntu 14.04 64-bit with KVM

Table 2: Local Hypervisor - Delft Brasserskade

Model	Dell PowerEdge R210 II
CPU	Intel®Xeon®CPU E3-1220L V2 @ 2.30GHz (Dual Core)
Memory	8GiB RAM DIMM DDR3 Synchronous 1333 MHz
Disk	Seagate ST1000NM0011 - 1TB (non SSD)
OS	Ubuntu 14.04 64-bit with KVM

Table 3: Remote Hypervisor - Amsterdam Science Park

4.3. Software tools

This section shows the tools used for this project and what their purpose is within the project.

Libvirt

"Libvirt [15] is collection of software that provides a convenient way to manage virtual machines and other virtualization functionality, such as storage and network interface management". This software collection includes an API library, a daemon (libvirtd), and a command line utility (virsh). The Libvirt software is a crucial part of the project, as its utilities are widely used in the Proof of Concept (see 5). This generic collection of software is not bound to a specific vendor, thus can be used with common hypervisors such as Xen and KVM.

The collection also includes a tool called "virsh", which can get information about Virtual Machines available on the hypervisor; it is also used to create snapshots, deploying and removing VMs.

QEMU

QEMU[16][17] is a generic and open source machine emulator and virtualizer. QEMU is a hosted VM monitor: it emulates CPUs through dynamic binary translation and provides a set of device models, enabling it to run a variety of unmodified guest operating systems. In this project it is used by KVM to run the virtual machines.

KVM

"KVM [2] is a full virtualization solution for Linux on x86 hardware containing virtualization extensions (Intel VT or AMD-V)".

Using KVM, one can run multiple virtual machines running unmodified Linux or Windows images. Each virtual machine has private virtualized hardware: a network card, disk, graphics adapter, etc... The tool used to provision the VM, Libguestfs, uses KVM

Libguestfs

"Libguestfs[18] is a library and has tools for accessing and modifying VM disk images." It can access most of the common disk images such as raw image files and "qcow2" (copy on write) image files, which are used in this project.

The tools used from libguestfs are the following:

- Python-guestfs is a python wrapper for the libguestfs tools. This wrapper allows to call the libguestfs function inside the python scripts we wrote in the Proof of Concept (PoC).

- virt-builder[19] allows to create new images; it can accept a variety of option in its command-line interface. By default, a minimal version of the chosen Operating System is retrieved from the Internet, then unpacked and provisioned in a new VM. Depending on the parameters, it can also install packages on the VM and run scripts after its first boot.
- virt-install[19] is another tool included in libguestfs. It is used to deploy and boot a VMs on the hypervisor.
- guestmount is a tool that allows to easily mount a VM or even a snapshot, eventually following the whole chain of snapshot dependencies. During analysis and the smart sync phase of the smart migrate 5 this tool is used.

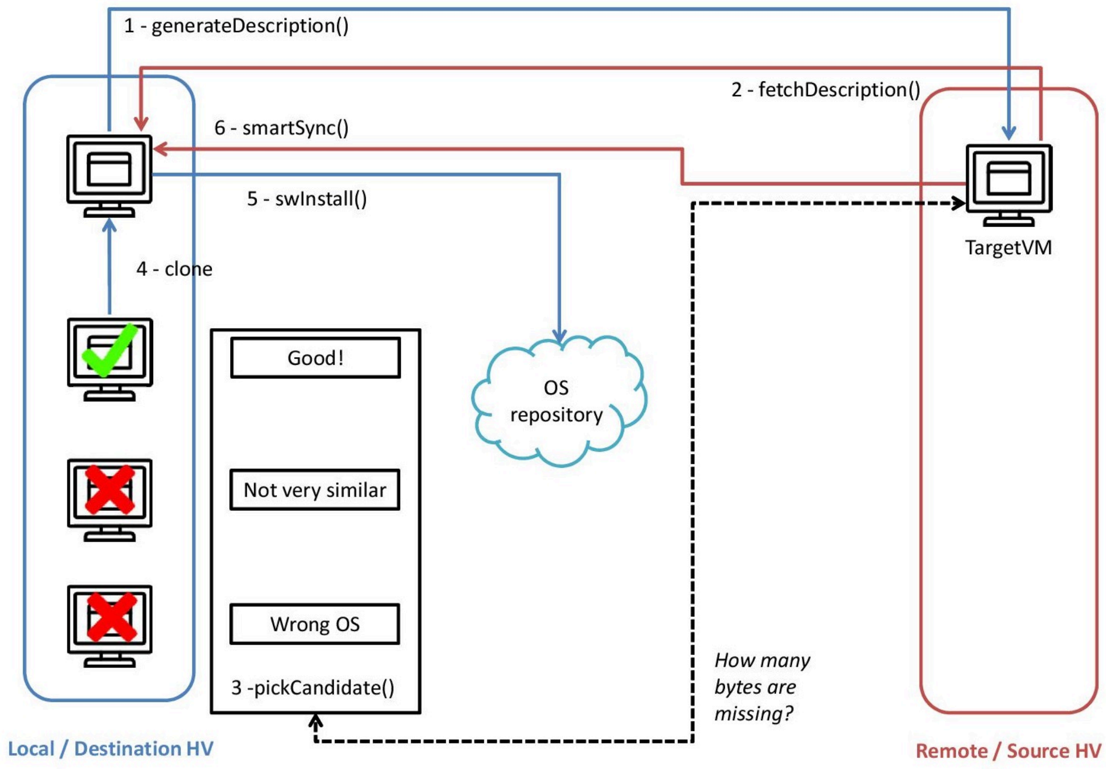

Figure 1: Smart Migrate algorithm scheme

5. The proof of concept Smart migrate

“Smart migrate” is the name of the script that teleports a Virtual Machine. As mentioned before, the aim of teleporting a VM is to try to reduce the amount of data sent between the source and the destination hypervisors.

5.1. Steps

“Smart migrate” is a bash script which calls a variety of tools and python scripts. Its algorithm (see also fig.1) is divided in the following steps:

Generate Description: contact the source hypervisor and ask it to generate the description of the target VM for the teleportation.

Fetch Description: download the generated description from the source hypervisor.

List Images: output a list of local VMs having the same Linux distribution and version.

Pick Candidate: find the best candidate for the teleportation from the list.

Create New Machine: create a new VM with a minimal installation.

Software Prepare: output a list of packages needed to be installed and/or removed from the new VM.

Software Install: download and install/remove the packages from the list.

Smart Sync: copy to the new VM any specific data which cannot be downloaded/recreated from another source. This data is referred as user data in this project.

Generate and Fetch Description

The generation and fetching of the description is an important part of the teleportation. The description shows properties of the Virtual Machine that will be teleported, such as the number of CPUs and network interfaces, as well as the list of distribution packages installed. This description is exported and transferred as a JSON file which can be easily parsed.

List Images

As described before, this step outputs a list of local Virtual Machine images having the same Operating System and version as the source VM.
It is worth to mention that this process also takes in consideration any snapshots of any VMs on the local hypervisor. It is safe to assume that the VM has only one OS installed, as this is a common practise amongst systems administrators that deals with VMs.

Pick Candidate

Pick Candidate uses a list of images to decide which image is the most suitable to use for the teleportation. It evaluates the image having the least amount of bits needed to be copied from the source VM, by using the well-known `rsync` program with its `--dry-run`[20] which, instead of a normal run, simulates a transfer and quickly outputs the result. This step can be perfected as it does not take in consideration the amount of packages that needs to be installed or removed.

Create New Machine

Create New Machine creates a new VM with the minimal installation of the same OS of the source VM. This step is run only if there is no suitable candidate found by 5.1. This new Virtual Machine will then act as precursor for the teleportation.

Software Prepare and Install

These two steps prepare the Virtual Machine for the installation/removal of the packages. In this phase any extra repository from the source VM and a special installation script are put on the new VM, which is booted in order to do the installation/removal (see 6.4). At the end of this process, the VM is shutted down.

Smart Sync

Smart Sync is the phase where the user data is transferred. This is done by using two different run of `rsync`. The first is the used to transfer all the data excluding the directories where the packages are installed.
After this, another `rsync` is run to sync all the files that might have been created by manual installation of compiled software (thus not available in the distribution repositories).

5.2. Experiments

To address the research questions the following experiments have been done. They have been chosen to show how well smart migrate might perform with different local VMs on the destination. From the worst case of having no local VM to use as a candidate, to the best case of having an almost identical VM available in the destination hypervisors. These are the cases chosen:

1. `rsync` of a raw disk image file (to simulate a normal migration).
2. `rsync` of a sparse disk image file (to simulate a normal migration).
3. Teleportation with no local VM available.
4. Teleportation with a local VM having a minimal installation.
5. Same situation as above, will all the packages updated to last version installed on the VM.
6. Same situation as above, but with also all software packages required by the source installed on the VM.
7. Same situation ad the previous experiment, but a different (non compatible) webserver is installed on the candidate VM
8. A local VM with the same software installed (including extra repository ones).

All experiments have been done with a minimal set of candidate images on the destination hypervisor. The source and destination servers are the same in each experiment. No bandwidth throttling has been enforced, the maximum download speed from the source to the destination is 10 MB/s (during our measurements it was pretty stable). Each experiment has been done five times to make an average.

In order to do timing measurement, the command `/usr/bin/time` has been run on each part of the teleportation software (see 5). With this the wall-clock, user and system time have been measured. The amount of data sent was also logged from the `rsync` commands called during the teleportation.

6. Results

6.1. Testing if the teleportation was successful

After teleporting the a VM a check was done to see if it was still functioning. After checking the VM it was confirmed that the websites (see table 1) were running correctly with the same content. The installed packages list was the same as the source, even the libraries, configuration and software manually compiled on it were present. ISPConfig was running and there was no error message displayed. Although we're still far from automatically determining if the teleportation always outputs a functional copy, from our tests it can be concluded that the VM was successfully teleported.

6.2. Bandwidth results

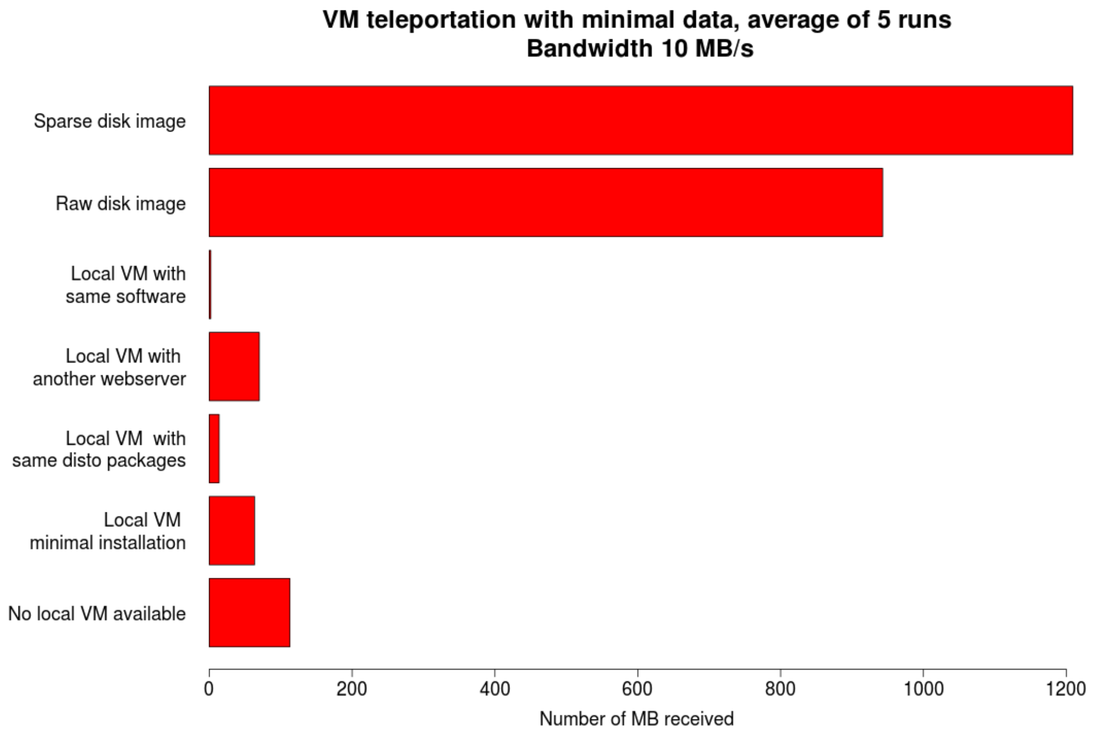

Figure 2: Teleportation of 2GiB VM - bandwidth consumption

Fig. 2 shows the results of VM standard copy and teleportation while having different local VMs on the destination hypervisor. By looking at fig. 2 the number of bytes between the host and client is greatly reduced. This is because the only thing that is send between the two parties is the difference in user data. Since the user data to migrate are configuration files and some small websites, the difference is great.

Needless to say, the more similar the destination VM gets to the source, the less the amount of data transferred, but in every case it is significantly less than the with the normal copies. More data is transferred for the sparse disk image because the latter consists of three snapshots that, apparently, brings some overhead.

For the user data only a few optimization can be done to decrease the amount of data needed to be send. All the information needs to be transferred and checked against the original data. The user data is all unique data and should be exactly copied. One noteworthy exception is the data that is stored in a database: in most cases, dumping this information to text file, transferring it compressed and having the destination restoring it, would probably decrease the amount of data transferred.

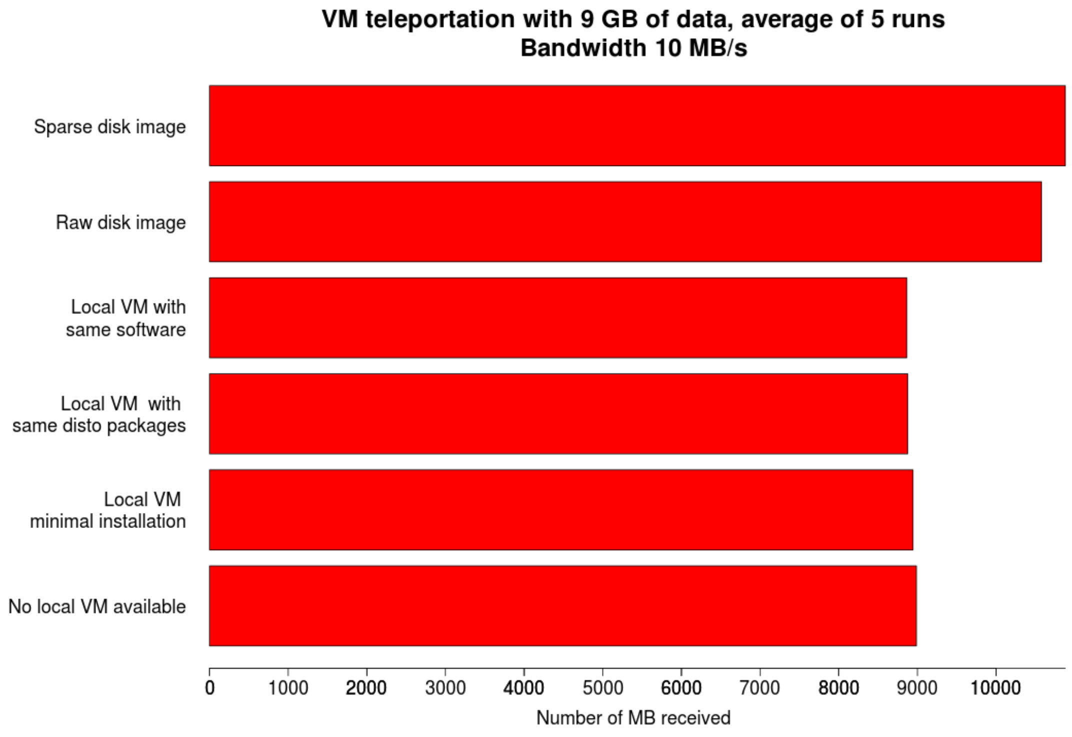

Figure 3: Teleportation of 11GiB VM - bandwidth consumption

Fig. 3 shows that with “large” amounts of user data the bandwidth saved with the smart migrate is not very big. This is because all the data that is being transferred is unique and the only source of that data is the remote VM. The only way to improve this is by having a local VM on the destination having at least partly the same data.

6.3. Time results

Fig.4 shows the results of the same tests as in Fig. 2, but from the time point of view. The time spend during syncing is correlated to the amount of user data on the source VM, and can only be decreased by improving the bandwidth or having similar local copies of the data. The less bandwidth between the two hypervisors, the faster the teleportation. In places where the bandwidth is not good this might be an option, also

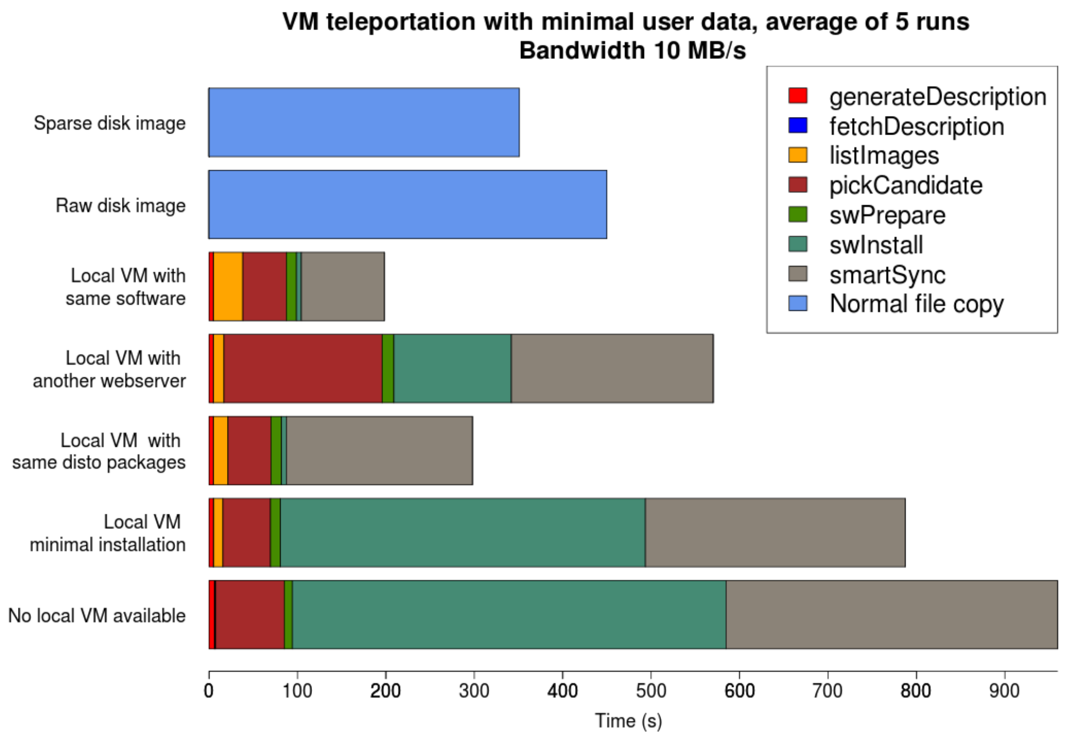

Figure 4: Teleportation of 2GiB VM - execution time

because of the distribution of common distribution repositories, taking not much time to download their compressed packages.

It is clear from looking at figure 4 that, as long as you have local VMs similar to the source VM, the time spend doing the teleportation is decreased. This shows that algorithm is working as intended. Another thing to notice is that the time spend doing the teleportation heavily depends on the amount of images where the candidate needs to be picked from. It may be noticed that the "Local VM with another webserver" bar has a far longer pick candidate phase than the other teleportation experiments: this is because in the local hypervisor there were three snapshots to analyze instead of just one(see section 4.2), the time spend is about three times as much compared to the other experiments. In this graph it is also clear to see that in some cases, namely the cases where there is no similar VM on the destination, just copying the machine will give a much better result than the teleportation. This shows that teleportation can be faster only in certain situations.

Fig.5 shows that there is a time benefit only with local VMs having mostly the same Operating System structure. Worth to mention is that all phases before smart sync take relatively less time as the user data increases. As there is no particular benefit with most situations here, it can't be stressed enough that the advantages of the teleportation heavily relies on having similar data locally available.

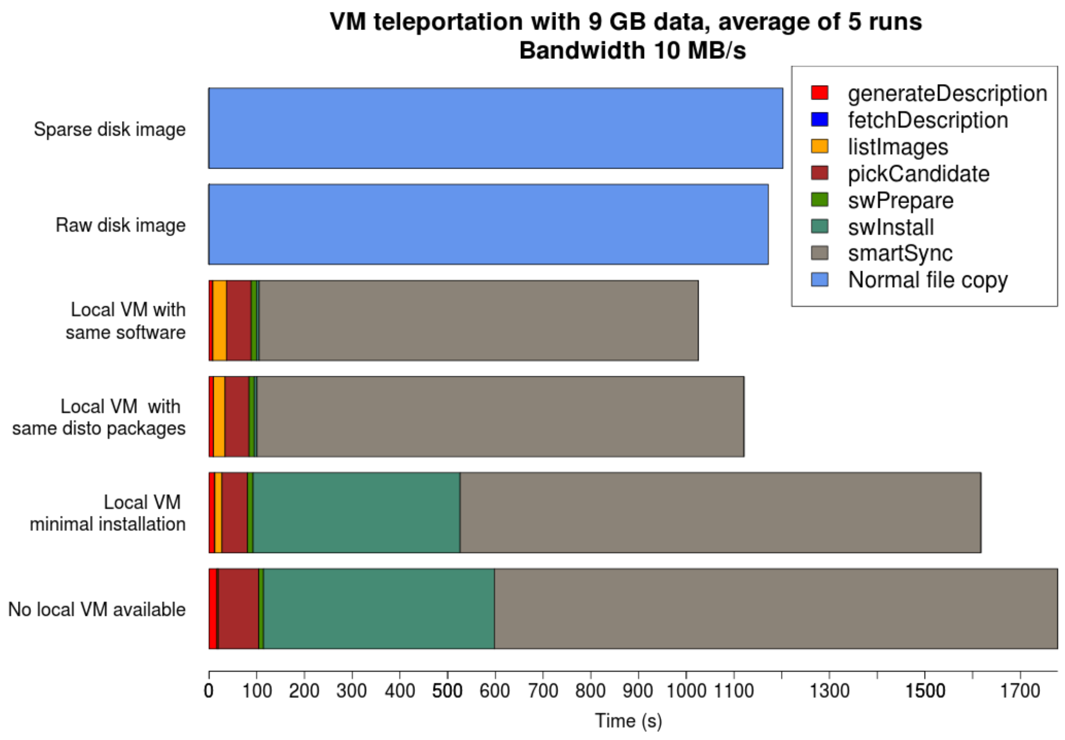

Figure 5: Teleportation of 11GiB VM - execution time

6.4. Problems

Following are the most significant problems encountered during the project.
A relevant one was installing packages on a powered off VM, which *should* be done effortlessly by the tool `virt-builder` (see 4.3). However, some packages apparently require the machine to be powered on, thus making impossible to install/remove all the packages without booting the VM (and slowing down the whole process). The suspicion is that some packages require services that are only available when the VM is booted, further analysis should be done. Installing packages on first boot did not give any problem, after having created some apposite scripts.

A problem similar to what described above was found, but this time the culprits were the repositories. Some of the packages that need to be installed might not be in the standard repos, but are available in third party ones (this often the case with CentOS, where very popular and maintained “extra” repositories are not uncommon to be installed). If these repositories are not installed the retrieval of some packages might fail. To solve this problem, the smart migrate algorithm takes care of copying any extra repository information and keys from the source to the destination VM before the installation script is run.

7. Conclusions

In our project the viability of VM teleportation has been tested. With the Proof of Concept it has been shown that there is a way to create a functional copy of a VM using VM teleportation.

From the results we can conclude that, in cases where there are very similar VMs on the destination VM, teleportation can even be faster than a normal copy. If there is no VM or a very minimal one however, a plain file copy is just faster. The results also show that bandwidth is saved in each experiment conducted. The amount of sent data depends on the difference in packages between the source VM and the candidate that is chosen on the destination. Even in the worst case scenario where there is no candidate available the amount of data send between the two parties is less.

This makes VM teleportation a viable way of transferring VMs between servers. This could be applied in applications like a VMDN, where multiple similar/same VMs might be transferred to the same destination.

8. Future work

The experiments that have been done were limited due to time constrains. Experiments need to be done with other Virtual Machines. Testing the algorithm in a real case scenario, with production VMs, would definitively give a better indication of how well the algorithm does and in which situations it will work better or worse.

We believe that the concept of "functional copy" is worth to receive a more formal/strict definition, as it is a crucial basis for further developing.

The algorithm should also be improved. One suggestion is about the installation of the software and the subsequent sync, which can be run in parallel as the former does not depend on the data being transferred during the sync. Also, the "pick candidate" phase can be improved: the script now mounts any possible candidate disk, one by one, they can instead be mounted at the same time and the `rsync` "dry runs" can be run in parallel. A threshold can also be put in place that if the image passes a certain threshold of bytes that have to be received (i.e. a bigger quantity comparing to the previously analysed candidate), then the algorithm immediately discard that image and save time. The algorithm can be smarten also by making it actually decide if the VM teleportation is worth doing, or if it should (e.g. if the bandwidth between two endpoints is excellent) just copy the virtual machine instead.

During this project only time and bandwidth consumed during the teleportation were considered. Further experiments might also take CPU and memory usage into account and see their impact on the hypervisors.

The Proof of Concept is, at the moment, a combination of Python and Bash scripts. An improvement would be writing everything in Python and make it more uniform. Also, the improved software should follow a client-server architecture, which is desirable in such environments.

A. Software tools

Table of software	
Software	Version
Libguestfs	1.24.5
KVM	2.0.0 (Debian 2.0.0+dfsg-2ubuntu1.10)
Kernel	3.13.0-44-generic
python-guestfs	1:1.22.7-1 (ubuntu package)
libguestfs-tools	1:1.24.4-1 (ubuntu package)
virsh	1.2.2

Table 4: These are the versions of the software used

B. Python code

The most important python functions used in for the smart migrate.

```
import guestfs
import jsonpickle
from collections import namedtuple
from subprocess import check_output
import parsers
import os
import re

from vm_info import VM_Info


def decode_description(description):
    f = open(description)
    vm_i = f.read()
    vm_i = jsonpickle.decode(vm_i)
    vm_i = namedtuple('VM_info', vm_i.keys())(*vm_i.values())
    return vm_i

def compare_descriptions(description1, description2):
    d1 = decode_description(description1)
    d2 = decode_description(description2)
    return difference_lists(d1.packages, d2.packages)

def difference_lists(l1, l2):
    add = list(set(l1).difference(l2))
```

```python
    # The packages in dest which are not in source
    remove = list(set(l2).difference(l1))
    return add, remove

def compare_package_lists_from_image(img1,img2):
    g = guestfs.GuestFS(python_return_dict=True)
    vm1 = inspect_vm(g, img1)
    g2 = guestfs.GuestFS(python_return_dict=True)
    vm2 = inspect_vm(g2, img1)
    diff = list(set(vm1.packages) - set(vm2.packages))
    print diff


def get_installed_packages(gfs, root):
    pkm = gfs.inspect_get_package_management(root)
    package_list = []
    if pkm == "yum":
        r_packages = gfs.sh("yum list installed")
        packages = str(r_packages).splitlines()
        # first 2 line are for the column information
        for p in packages[2:]:
            #tmp += p.split(" ")[0] + ","
            package_list.append(p.split(" ")[0])
    elif pkm == "apt":
        # The same as doing virt_insepctor returns a dict.
        packages = gfs.inspect_list_applications2(root)
        for p in packages:
        #    tmp += app["app2_name"] + ","
            package_list.append(p["app2_name"])
    else:
        print "Unknown package manager"
    return package_list

def virt_builder_command(vm_i, output):
    command =  "virt-builder %s-%s -o %s --format qcow2 --root-password " \
         "password:password --hostname %s  --size %sb" % (vm_i.distro,vm_i.version,  output, vm_i.hostname, vm_i.size)
    return command

def inspect_vm(disk):
    g = guestfs.GuestFS(python_return_dict=True)
    try:
        g.add_drive_opts (disk, readonly=1)
    except:
        print "Image does not exist"
        return
    # Run the libguestfs back-end.
    g.launch ()
    roots = g.inspect_os ()
    if len (roots) == 0:
        Exception("No os found")

    # TODO need to find a good way to inspect multiple vm's at the same time.
    for root in roots:

        ma_v = g.inspect_get_major_version (root)
        mi_v = g.inspect_get_minor_version (root)
        os_type = g.inspect_get_type (root)
        distro = g.inspect_get_distro (root)
        pkm = g.inspect_get_package_management(root)
        hostname = g.inspect_get_hostname(root)
        # command give a new line. It is removed by removing the last two lines.

        size = check_output(['qemu-img', 'info', disk])
        p = re.compile(r"\d* bytes")
        size =  p.findall(size)[0][:-6]

        vm_i = VM_Info(ma_v, mi_v, distro, os_type, pkm, hostname, size)

        # Sort keys by length shortest first, so that we end up

        # mounting the filesystems in the correct order.
        mps = g.inspect_get_mountpoints (root)
        def compare (a, b): return len(a) - len(b)
        for device in sorted (mps.keys(), compare):
            try:
                g.mount (mps[device], device)
            except RuntimeError as msg:
                print "%s (ignored)" % msg

        packages = get_installed_packages(g, root)
        # add packages to the vm info
        vm_i.packages = packages
        encoded = jsonpickle.encode(vm_i, unpicklable=False)
        #TODO chose another path for the description to be made.
        f = open("./" + ".".join(disk.split("/")[-1].split(".")[:-1])
            +".description", "w")
        f.write(encoded)
        f.close()
         # Unmount everything.
        g.umount_all ()
        return vm_i

def inspect_vm_domain(domain):
    g = guestfs.GuestFS(python_return_dict=True)
    g.add_domain(domain, readonly=1)
    # Run the libguestfs back-end.
    g.launch ()
    roots = g.inspect_os ()
    if len (roots) == 0:
        Exception("No os found")

    # TODO need to find a good way to inspect multiple vm's at the
        same time.
    for root in roots:

        ma_v = g.inspect_get_major_version (root)
        mi_v = g.inspect_get_minor_version (root)
        os_type = g.inspect_get_type (root)
        distro = g.inspect_get_distro (root)
        pkm = g.inspect_get_package_management(root)
        hostname = g.inspect_get_hostname(root)
        # command give a new line. It is removed by removing the last
             two lines.
        size = check_output(['stat', '-c', '%s', domain])[:-1]
        inst_info = parsers.parse_install_info(domain)

        vm_i = VM_Info(ma_v, mi_v, distro, os_type, pkm, hostname,
            size)

        # Sort keys by length shortest first, so that we end up

        # mounting the filesystems in the correct order.
        mps = g.inspect_get_mountpoints (root)
        def compare (a, b): return len(a) - len(b)
        for device in sorted (mps.keys(), compare):
            try:
                g.mount (mps[device], device)
```

```python
            except RuntimeError as msg:
                print "%s (ignored)" % msg
        packages = get_installed_packages(g, root)
        # add packages to the vm info
        vm_i.packages = packages
        encoded = jsonpickle.encode(vm_i, unpicklable=False)
        #TODO chose another path for the description to be made.
        f = open("./" + domain +".description", "w")
        f.write(encoded)
        f.close()
         # Unmount everything.
        g.umount_all ()
        return vm_i



def deploy_vm_command(vm_i, name):
    ram = 2048
    name = ".".join(name.split(".")[:-1])
    command = "virt-install --name %s --ram %d --disk %s --import" % (vm_i.hostname, ram , name )
    # command = "virt-install --name %s  --disk %s --import" % (vm_i['hostname'], name )
    f = open("virt_install_script.sh",'w')
    f.write(command)
```

The code for listing the VM and comparing descriptions.

```python
import parsers
import utils


def find_snapshots_for_domain(domain):
    return check_output(["virsh", "snapshot-list", domain, "--name"]).split("\n")

def find_all_domains():
    return check_output(["virsh", "list", "--all", "--name"]).split("\n")

def get_snapshot_xml(dom, snap):
    ret = check_output(["virsh", "snapshot-dumpxml", dom, snap])
    # print ret
    return ret
def get_domain_xml(dom):
    ret = check_output(["virsh", "dumpxml", dom])
    # print ret
    return ret


def get_all_images():
    s = set()
    # dump all stuff
    for dom in find_all_domains():
        if dom == "":
            continue
        for snap in find_snapshots_for_domain(dom):
            if snap == "":
                continue
            # get the image path for a snapshot
            s.add(parsers.get_file_for_snapshot(get_snapshot_xml(dom
                ,snap)))

        s.add(parsers.get_file_for_domain(get_domain_xml(dom)))
    return s

def list_all_images_compared(src):
    src_info = utils.decode_description(src)
    images = get_all_images()
    ret = ""
    # print src_info.version, src_info.distro
    list_remove = set()
    for image in images:
        try:
            info = utils.inspect_vm(image)
        except:
            print"File %s could not be inspected" % image
            list_remove.add(image)
            continue
        # print info.distro,info.version
        if info.distro != src_info.distro or info.version != src_info
            .version:
            list_remove.add(image)
    images = images.difference(list_remove)
    if  images:
        ret = "\n".join(images)
    return ret

def vm_compare(src):
    dest_info = {}
    src_info = utils.decode_description(src)
    # make all the descriptions
    list_ar = {}
    for image in get_all_images():
        temp_info = utils.inspect_vm(image)
        # Check if this image is relevant.
        if temp_info.distro == src_info.distro and temp_info.version
            == src_info.version:
             dest_info[image] = temp_info
    # make the difference between.
    for image, info in dest_info.iteritems():
        add, remove = utils.difference_lists(src_info ,info)
        list_ar[image] = [add,remove]
    return list_ar
```

C. Bash code

`rsync` command to analyze differences between two Virtual Machines:

```bash
HYPERVISOR=$1
R_MNT_PATH=$2
L_MNT_PATH=$3

rsync -azAXn --delete --stats --exclude={"/dev","/tmp","/proc","/sys","/var/tmp","/run","/mnt","/media","/lost+found","/usr","/lib","/etc/fstab","/lib32","/lib64","/boot"} $HYPERVISOR:$R_MNT_PATH/ $L_MNT_PATH
```

`rsync` commands to missing data from source to destination VM:

```bash
HYPERVISOR=$1
R_MNT_PATH=$2
L_MNT_PATH=$3

rsync -azAX --delete --stats --exclude={"/dev","/tmp","/proc","/sys","/var/tmp","/run","/mnt","/media","/lost+found","/usr","/lib","/etc/fstab","/lib32","/lib64","/boot"} $HYPERVISOR:$R_MNT_PATH/ $L_MNT_PATH
rsync -azAX --ignore-existing --stats --exclude={"/dev","/tmp","/proc","/sys","/var/tmp","/run","/mnt","/media","/lost+found","/boot"} $HYPERVISOR:$R_MNT_PATH/ $L_MNT_PATH
```

Script for generating random data:

```bash
#!/bin/bash

rm -rf /root/TEST 2> /dev/null
mkdir /root/TEST
cd /root/TEST
for i in `seq 1 10`; do dd if=/dev/urandom of=512MB_$i bs=64M count=8; done
for i in `seq 1 10`; do dd if=/dev/urandom of=256MB_$i bs=64M count=4; done
mkdir 1mega
dd if=/dev/urandom of=masterfile bs=64M count=4
split --bytes=1M masterfile 1mega/1mb
mkdir 256kilo
dd if=/dev/urandom of=masterfile bs=64M count=8
split --bytes=256KB masterfile 256kilo/256kb
rm -f masterfile
```

References

[1] Maarten Fonville. *The Virtual Machine Delivery Network*. 2014.

[2] *Kernel Based Virtual Machine*. URL: http://www.linux-kvm.org/page/Main_Page (visited on 01/07/2014).

[3] *Data cluster*. URL: http://en.wikipedia.org/wiki/Data_cluster (visited on 01/09/2014).

[4] Franco Travostino et al. "Seamless live migration of virtual machines over the MAN/WAN". In: *Future Generation Computer Systems* 22.8 (2006), pp. 901–907.

[5] *Performing VM migration under Xen*. URL: http://wiki.xen.org/wiki/Migration (visited on 02/03/2014).

[6] *vMotion*. URL: http://www.vmware.com/products/vsphere/features/vmotion (visited on 02/03/2014).

[7] *XenServer migrate machines between hosts*. URL: http://serverfault.com/questions/396116/xenserver-migrate-machines-between-hosts (visited on 02/03/2014).

[8] *Creating virsh dump files*. URL: https://access.redhat.com/documentation/en-US/Red_Hat_Enterprise_Linux/6/html/Virtualization_Administration_Guide/sect-vish-dump.html (visited on 02/03/2014).

[9] *vCenter Converter*. URL: http://www.vmware.com/it/products/converter (visited on 02/03/2014).

[10] *Puppet Homepage*. URL: http://puppetlabs.com/ (visited on 01/07/2014).

[11] *Vagrant Homepage*. URL: https://www.vagrantup.com/ (visited on 01/07/2014).

[12] *Welcome to Python.org*. URL: https://www.python.org (visited on 02/07/2014).

[13] *Bash - GNU Project - Free Software Foundation*. URL: www.gnu.org/software/bash (visited on 02/07/2014).

[14] *ISPConfig*. URL: http://www.ispconfig.org/page/home.html (visited on 02/03/2014).

[15] *Libvirt: The virtualization API*. URL: http://libvirt.org/ (visited on 01/07/2014).

[16] *QEMU*. URL: http://wiki.qemu.org/Main_Page (visited on 01/26/2014).

[17] *QEMU - Wikipedia, the free encyclopedia*. URL: http://en.wikipedia.org/wiki/QEMU (visited on 01/27/2014).

[18] *libguestfs, library and tools for accessing and modifying VM disk images*. URL: http://libguestfs.org/ (visited on 01/26/2014).

[19] *virt-builder*. URL: http://libguestfs.org/virt-builder.1.html (visited on 01/26/2014).

[20] *rsync - Linux man page*. URL: www.gnu.org/software/bash (visited on 02/07/2014).