Politecnico di Torino

Introduction to Computing Virtualization

Fulvio Risso

October 20, 2021

License

Acknowledgments

The author would like to thank all the people who contributed to this document.

Contents

Part I.

Introduction

1. Introduction

This lab aims at installing a new Operating System from scratch in a virtual machine (VM) running within a computing virtualization framework. In this respect, this lab requires to interact with two different hypervisors:

- **Oracle Virtualbox**[1]: more oriented to occasional/non-professional users, freely available for most operating systems, with great support for GUI-based guest OS (e.g., Windows);
- **Linux KVM**[2]: more oriented to production-grade virtualization, available only in Linux, targeting preferably console-based guests such as Linux servers, although GUI-enabled Guest OSes can be supported as well.

Upon installation, the VM will be cloned and modified in its hardware components, showing how virtualization allows to quickly update the (virtual) hardware to match new necessities.

To motivate the change in the hardware, this lab simulates a scenario in which two hosts belong to different LANs can exchange traffic through a router, connected to the same switched network of the previous hosts. This requires an update of the hardware definition of the installed VMs in order to create a router with two network interfaces.

IMPORTANT: due to timing limitations, all the lab related to Virtualbox SHOULD NOT BE CARRIED OUT IN THE LAB . This part is left to the student, for his individual study at home.

EXTRA: In addition to this text, we provide several videos which discuss some parts of the topics of this Lab. You can find them at

- `https://www.youtube.com/playlist?list=PLTAfidx4guQImT5beuAs4YAhIzuBBoEHk`

[1] `https://www.virtualbox.org/`
[2] `https://www.linux-kvm.org/`

Part II.

Oracle Virtualbox

2. Create a new VM and installing a new Guest OS

This part aims at creating a new VM, installing an Ubuntu Desktop operating system (as *Guest OS*) in it. This lab assumes that the official installation image of the above operating system ISO[1] has been already downloaded on your machine and hence is available locally.

Suggestion: as multiple versions of the above operating system are available, we suggest to select Ubuntu Desktop 19.04 in order to facilitate the interactions with the professor.

The creation of a new running virtual machine can be achieved through the following steps.

1. Create a new hardware VM, using the standard settings in Virtualbox.

   Suggestion: check that you have enough space on the physical disk (∼10GB), otherwise you may experience errors later. If the disk looks full, select another physical disk to store the disk image of your guest VM.

2. Customize the hardware by connecting the ISO image of Ubuntu Desktop (containing the image of the operating system to be installed) to the optical drive controller.

3. Start the virtual machine and install the new OS, answering to the questions required to set it up properly.

   Suggestion (1): install the OS in "minimal" mode, avoiding rich applications (e.g., LibreOffice, etc) that are not required in this lab.

   Suggestion (2): select the English language (or at least latin characters) in order to facilitate the interaction with the professor.

4. The installation process should take about 15 minutes; at the end of the process, you should verify that the guest OS works as expected by launching some applications in it.

2.1. Installing the "Guest Additions"

Now that you have the VM up and running, we strongly suggest to install the *VM Guest Additions*, which include additional drivers to be installed in the guest OS that facilitate the interaction with the user. For instance, capabilities such as to *resize* the display, or to copy text from the host to the VM and vice versa (i.e., a *shared clipboard*) work better if the Guest Additions are active in your system.

The installation of the *VM Guest Additions* can be achieved with the following steps:

1. Select the menu "Devices" in your VirtualBox window (the one with your VM running in it), then "Insert Guest Additions CD Image...". A new window will appear asking you if you want to run the executable that has been found on the CD-ROM.
2. Confirm that you want to run the executable; a new window may appear asking for the admin password (which corresponds to *your* password), then a terminal will appear that shows the progress of your install.

[1] `http://releases.ubuntu.com/`.

3. When the installation finishes, you may be asked to reboot the VM to have the Guest Additions working.

After rebooting, you can see that the VM window can be resized smoothly, and that the shared clipboard works (this requires one further step, going to the menu "Devices", then "Shared Clipboard", then select "Bidirectional").

2.2. Connect to the other machines in the lab

This part can be done only if your lab includes a physical network and the lab owner assigns IP addresses to any host in the lab (even virtual machines).

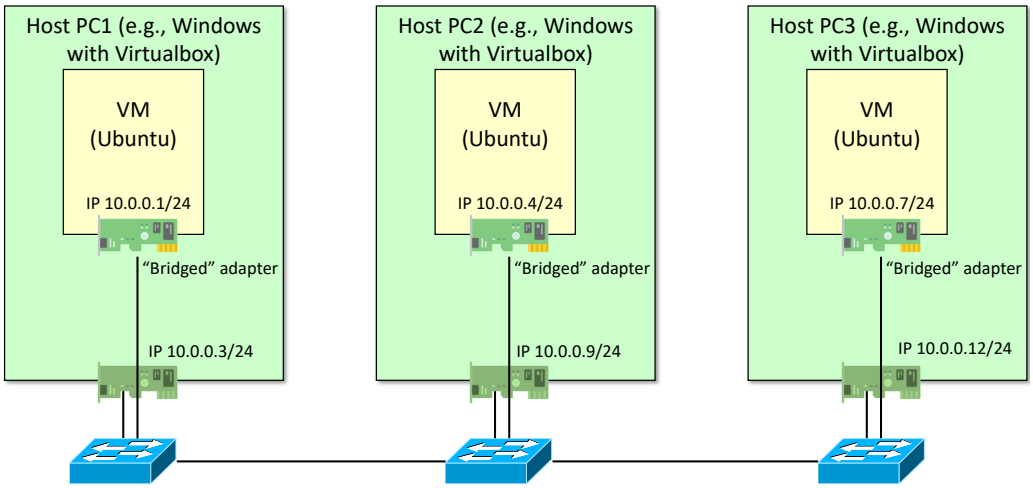

Figure 2.1: VMs are connected in bridged mode to the physical lab network.

1. Turn the machine off (e.g., click the *shutdown* button in the VM). Changes to the hardware of the machine can usually be done only when the VM is not running.
2. In Virtualbox, go the network settings of the VM, select the network interface, locate the "*Attached to*" box and select "*Bridged Adapter*". In this case the VM looks like directly connected to the physical network, as shown in Figure 2.1: the network will see *six* hosts, where *three* are in fact physical hosts, and *three* are VMs. Note that the network infrastructure cannot notice any difference between physical hosts and VMs.

   In this case, your VM and the your host PC will have different IP addresses, all belonging to the same IP network, such as in Figure 2.1.
3. Ask for the IP address active in other VMs in your lab, e.g., the IP address of the VM of one of your friend. The IP address of your interface can be see by typing `sudo ip addr`, which produces an output that looks like the following:

```
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group
        default qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
    inet6 ::1/128 scope host
       valid_lft forever preferred_lft forever
```

```
2: enp0s3: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc fq_codel state UP group
        default qlen 1000
    link/ether 08:00:27:c0:81:04 brd ff:ff:ff:ff:ff:ff
    inet 10.0.0.1/24 brd 10.0.0.255 scope global dynamic noprefixroute enp0s3
       valid_lft 85981sec preferred_lft 85981sec
    inet6 fe80::b940:2258:137f:7fdb/64 scope link noprefixroute
       valid_lft forever preferred_lft forever
```

4. Supposing that your friend is on *PC3*, hence the IP address of his VM is `10.0.0.7` (according to Figure 2.1), *ping* the VM of your friend from inside your virtual machine, e.g., by typing `ping 10.0.0.7`.

   If everything works fine, you will see an output similar to the following, which means that the *ping* succeeded and that the network works fine:

```
PING 10.0.0.7 (10.0.0.7) 56(84) bytes of data.
64 bytes from 10.0.0.7: icmp_seq=1 ttl=64 time=0.119 ms
64 bytes from 10.0.0.7: icmp_seq=2 ttl=64 time=0.178 ms
64 bytes from 10.0.0.7: icmp_seq=3 ttl=64 time=0.172 ms
64 bytes from 10.0.0.7: icmp_seq=4 ttl=64 time=0.174 ms
```

2.3. Different types of network connectivity in Virtualbox

Virtualbox supports for your VMs multiple types of network options. This enables the creation of VMs that can tale to each other but cannot connect to the Internet (e.g., attached to an "*internal network*"), other that cannot talk to each other but connect to the Internet (e.g., attached to "*NAT*"), other that look like directly attached to the physical network (e.g., attached to a "*bridged network*" as before), and more.

More details are available at `https://www.thomas-krenn.com/en/wiki/Network_Configuration_in_VirtualBox`; Figure 2.2 shows a summary of the expected behavior.

Network type	Access Guest -> other Guests	Access Host -> Guest	Access Guest -> external Network
Not attached	-	-	-
Network Address Translation (NAT)	-	-	✓
Network Address Translation Service	✓	-	✓
Bridged networking	✓	✓	✓
Internal networking	✓	-	-
Host-only networking	✓	✓	-

Figure 2.2: Network options in Virtualbox.

3. Clone a VM and connect two VMs on a bridged network

This part aims at creating two VMs connected to an internal (bridged) network such as in Figure 3.1.

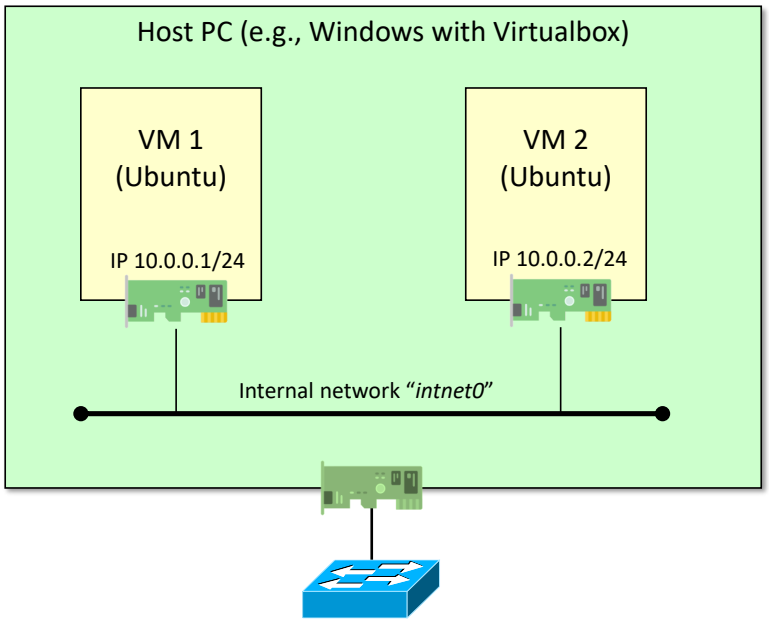

Figure 3.1: Two VMs connected to a bridged network, without Internet connection.

1. Clone the VM, paying attention to change the MAC address on the second VM. In fact, duplicated MAC addresses are not allowed on bridged networks such as in this setup (while they work properly if VMs are connected to in different LANs).

   Suggestion: use "*expert mode*" cloning, which enables to select the "*linked clone*" option in order to avoid copying the entire hard disk. Quoting from VMware documentation[1]:

   > **Full-clone**: A full clone is an independent copy of a virtual machine that shares nothing with the parent virtual machine after the cloning operation. Ongoing operation of a full clone is entirely separate from the parent virtual machine.
   >
   > **Linked-clone**: A linked clone is a copy of a virtual machine that shares virtual disks with the parent virtual machine in an ongoing manner. This conserves disk space, and allows multiple virtual machines to use the same software installation. Linked clones have their own virtual disk but this is just a differential disk that contains new files or whatever that the parent virtual machine didn't have when the snapshot of that linked clone was created.
   >
   > In case of full-clone, VMs are completely independent each other. Instead, linked clones always need the parent virtual disk as their base-virtual-disk.

[1] `https://www.vmware.com/support/ws5/doc/ws_clone_typeofclone.html`

2. From the Virtualbox console, modify the network settings of both machines by attaching their Network Interface Card (NIC) to an "Internal network" named *intnet0*. This is a new virtual network that connects both VMs and nothing else; hence, this way our VMs cannot connect to the Internet, as presented in Section 2.3.
3. Launch both VMs, open a command line terminal in them, and disable the "*network-manager*" service, which may overwrite the static configuration of IP addresses that we will complete in the next step:

   `sudo service network-manager stop`
4. List the name of your local interfaces with the command "`sudo ip addr show`"; you should get an output similar to the following:

```
1: lo0: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group
        default qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
    inet6 ::1/128 scope host
       valid_lft forever preferred_lft forever
2: enp0s3: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc fq_codel state UP group
        default qlen 1000
    link/ether 08:00:27:c0:81:04 brd ff:ff:ff:ff:ff:ff
```

   This lab assumes that the Ethernet network interface is named *enp0s3*; if the interface has a different name, change the following commands accordingly.
5. Now, set manually the IP address of the VMs with the following commands (please use the correct data, in Italic below, each one in the correct VM):

   - VM1: `sudo ip addr add 10.0.0.1/24 dev` *enp0s3*
   - VM2: `sudo ip addr add 10.0.0.2/24 dev` *enp0s3*

   Note: If you do not know the name of the network card in your VM, simply type "`sudo ip addr`" to show available network interfaces.
6. Now, verify that the two machines can *ping* each other:

   - Inside VM1: `ping 10.0.0.2`
   - Inside VM2: `ping 10.0.0.1`

   Suggestion: while pinging, open another terminal window and launch a network sniffer with `sudo tcpdump -n`: it will show the packets that are actually exchanged between both machines, such as ARP and ICMP.

4. Clone (again) and connect two VMs on a routed network

This part aims at creating a more complex topology that includes three VMs, connected to *two* internal (bridged) networks; one of the three VMs will act as an IP router, such as in Figure 4.1.

1. Starting from the previous part, clone one more time the original VM. However, in this case we should pay more attention to the network configuration:
   - VM1 and VM3 will have only one NIC, as before, although they are attached to different internal networks (respectively *intnet0* and *intnet1*)
   - VM2 needs to have two NICs, attached to different internal networks (respectively *intnet0* and *intnet1*); therefore, we have to modify the hardware configuration of the VM and add a new NIC.
2. Start all the three VMs and configure the proper IP address on each interface, using the commands already shown before.

   Suggestion (1): if you get confused about which VM are you in, you can change the prompt of the command line terminal e.g., by typing `PS1="VMname> "` where "`VMname`" can be any meaningful string you like.

   Suggestion (2): for VM2, pay attention which interface is attached to which network.

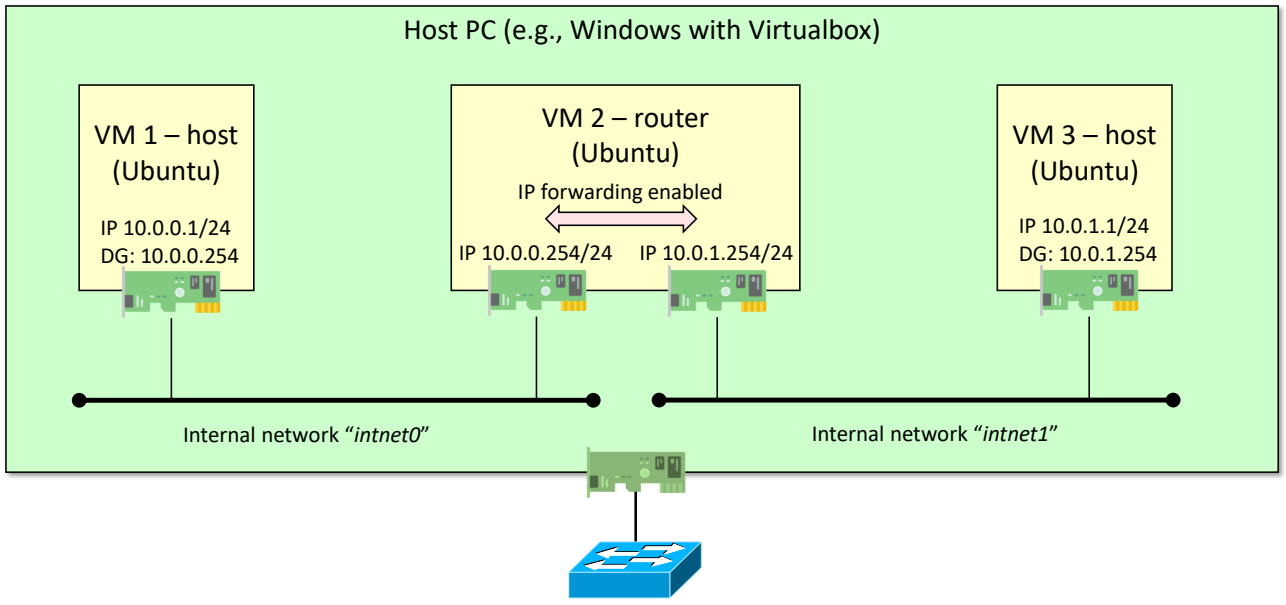

Figure 4.1: Two VMs connected through a (third) router VM, without Internet connection.

3. You can now verify that VM1 pings VM2, and that VM2 pings VM3. However, VM1 and VM3 do not ping each other. There are two reasons for this misbehavior, which relate to an incomplete network configuration:

   - Neither VM1 nor VM3 have their default gateway set. Therefore, they cannot reach destinations that belong to remote IP networks.
   - VM2 is not configured to act as a router; therefore, it cannot receive packets from one NIC and forward them on another, which is what VM2 is supposed to do in this lab.

4. Set the default gateway in VM1 and VM3, which would be the corresponding IP address of VM2 in the above nentworks:

   - Type the following command on VM1:

     `sudo ip route add default via 10.0.0.254`

   - Verify that the default gateway is now properly configured:

     `ip route`

   - Repeat the same commands (with the proper values) inside VM3.

5. Enable IP routing on VM2:

   `sudo sysctl -w net.ipv4.ip_forward=1`.

   Note: the `sysctl` command is used to change dynamically the behavior of the Linux kernel. Type `sysctl -a` to see the long list of variables that can be customized.

6. Check that the routing table is correct and that there are entries for both `10.0.0.0/24` and `10.0.1.0/24`:

   `ip route`

7. Now, verify that the PING works also between VM1 and VM3.

   (inside VM1): `ping 10.0.1.1`

Part III.

Linux KVM

5. Create a new VM with a pre-installed Ubuntu cloud image

While KVM supports the creation of a new VM in the same way as we saw in Section 2 (e.g., downloading an ISO image and installing the guest OS from scratch), this lab suggest to practice with another option that is much more common in cloud environment: setting up an already installed VM in KVM. In fact, often system administrators start from an already installed VM and then customize it according to their needs instead of installing a GuestOS from scratch, which requires more time.

For this reason, major Linux distribution already provide pre-installed VM targeting the cloud environment, such as in Ubuntu:

- **Ubuntu Cloud Images** (`https://cloud-images.ubuntu.com/` are the official Ubuntu images and are pre-installed disk images that have been customized by Ubuntu engineering to run on public clouds that provide Ubuntu Certified Images, Openstack, LXD, and more.
- **Ubuntu Minimized Cloud Images** (`https://cloud-images.ubuntu.com/minimal/`) are official pre-installed disk images that have been customized by Ubuntu engineering to have a small runtime footprint in order to increase workload density in environments where humans are not expected to log in. More information at `https://ubuntu.com/blog/minimal-ubuntu-released`.

Note (1): this lab will use the *Ubuntu Cloud Images*.

Note (2): installation instructions for KVM can be found in Appendix 6; here we just assume KVM (and the additional modules such as the `virt-manager` GUI) are already set up in your environment.

5.1. Physical and logical setup

This lab requires a physical setup such as depicted in Figure 5.1. Each student will be provided with a couple of VMs running in the POLITO datacenter on the CrownLabs platform (`https://crownlabs.polito.it`, one acting as Client and one representing our **server**.

The Client VM represents our **management host**, which is used to configure our (remote) server environment.

> **Note**: The case in which a **management host** connects and configures a **remote server** is really common in the real world; access to the server via console is rather uncommon because of the necessity for the system administrator to physically reside in the datacenter.
>
> However please note that the system admin must be very careful with the network configuration of the server: if some wrong commands are issued, you may loose network connectivity to the server, hence the possibility to configure the machine.
>
> This is the reason why often servers in a datacenter are provided with a parallel network that is used to *remotize* the access to the server console, to be used either in case of emergency or for the initial setup of the machine.

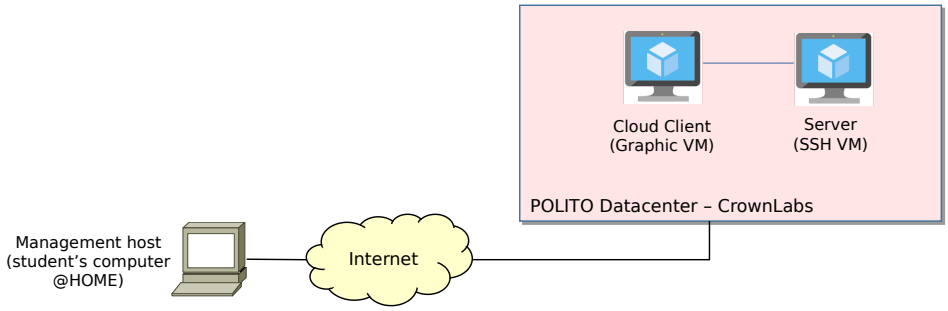

Figure 5.1: Physical setup: how to connect to your virtualized resources.

On the **server** side, this lab expects to create the configuration depicted in Figure 5.2: two VMs connected to a L2 virtual switch (Linuxbridge), which is then connected to the Linux stack and from there to the Internet. KVM will insert all the required NAT rules and routing table in order to allow VM1 and VM2 to connect to the Internet.

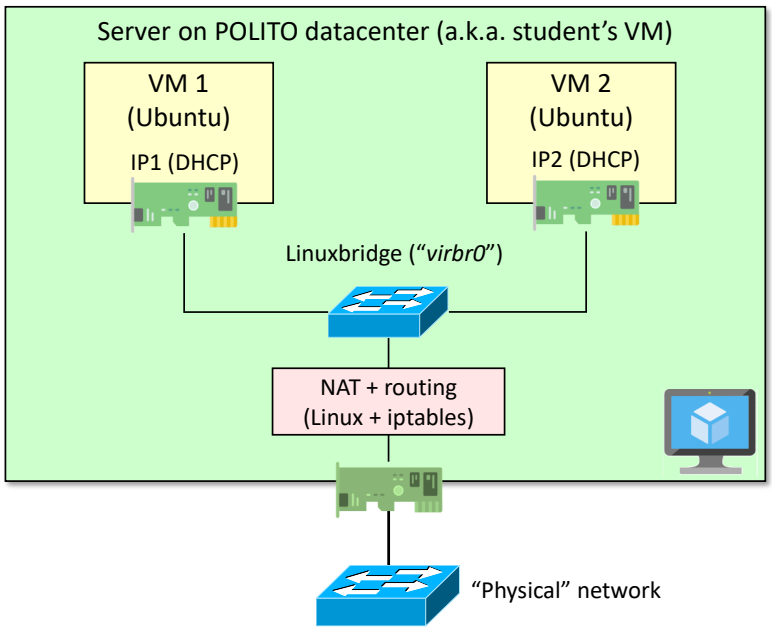

Figure 5.2: Logical setup: configuration that is expected in the target server at the end of the lab.

Please note that, being our server a virtual machine, this lab will use the concept of *nested virtualization*, as we will create a VM inside another VM.

5.2. Prepare the lab environment

5.2.1. Client VM (a.k.a. management host)

The **management host**, is used to control the remote hypervisor, running on the remote server.

This machine will use the SSH tool to connect to the remote server, and it will also exploit the GUI-based Virtual Machine Manager (`virt-manager`), which makes easier the interactions with the remote hypervisor.

In fact, while the KVM hypervisor can be controlled by using command-line tools (e.g., `virsh`, to create VMs, and more), this is not that easy at the beginning[1]. Instead, `virt-manager` provides a graphical interface (hence, cannot be directly executed on the server) that can control a remote hypervisor (hence, the VMs running on it) by issuing the proper commands through an SSH connection. Hence, `virt-manager` facilitates the interactions with the remote hypervisor that works in console-mode only (no GUI).

5.2.2. Laboratory VM (a.k.a. student's server)

The server comes with all the required tools preinstalled, hence nothing has to be done in order to prepare this environment.

Access to the server

First, you have to get the IP address of the server, the KVM Virtual Machine from the CrownLabs dashboard. The access to the server is possible with the following command (**on your management host**):

```
ssh netlab@IP_of_KVM_VM
```

If you are successfully authenticated, let's configure a proper SSH key pair. Using Key pairs relies on asymmetric authentication and it is normally a preferable way to connect to an SSH server. This is particularly interesting because you can disable password authentication and the possibility of dictionary attacks, which are a major threat if yuor host is exposed on the Internet.

In addition, key-based authentication can be secure but also completely not-interactive. This opens the door to completely automated configuration of the host without the need to install a dedicated agent and is used by very popular tools such as Ansible.

5.3. Configure proper storage folders on the server

IMPORTANT: unless explicitly specified, all the instructions below apply to the server. Hence, this lab assumes that the student is already logged in the server.

In order to create new VMs, we may need to store a set of ISO images (i.e., OS install disks, or configuration disks that will be used by `cloud-init`) and a set of VM images (i.e., the actual installed VM image) on the KVM server.

To avoid polluting your system, you should create two new directories, one to keep your ISO (install) disks, and one for your storage pools (installed VM images). The default storage pool in KVM is `/var/lib/libvirt/images`; we suggest to create instead two distinct folders under your home directory: `~/kvm-iso` and `~/kvm-pool`, as follows:

```
/home/netlab/
         |
         +----/kvm-iso
         |
         +----/kvm-pool
```

[1] Remember that servers usually do not have any graphical interface, hence all the interactions must be carried out using console-mode operations.

5.4. Download the Ubuntu Minimal Cloud image disk

Open a terminal and download the disk image of your choice, e.g., the latest LTS release, from the Ubuntu Minimized Cloud Images website (e.g., `wget https://cloud-images.ubuntu.com/focal/current/focal-server-cloudimg-amd64.img`).

- **Save** this image in your home folder;
- **Copy** the image into folder `~/kvm-pool`, while keeping the original in your home folder (you may need this file later).

5.5. Customize the disk image with your personal data

Cloud images are operating system templates and every instance starts out as an identical clone of every other instance. Before running the instance, images may need to be customized, e.g., with some additional software packages, by modifying some configuration parameter (e.g., network), and by creating the proper credentials for login[2].

In our lab you need simply to enable a user account in order to be able to login. We present here three possible procedures: the first two using `cloud-init`, the third using `virt-customize`.

5.5.1. Cloud-init (preferred)

`Cloud-init`[3] (`https://cloud-init.io/`) is one of the most widely used tool for passing some initialization parameters to a VM at boot time. `Cloud-init` relies on two text files that are read either from a predefined network location, or (such as in this lab) from a disk drive that is mounted to the VM at boot, such a CD-ROM device.

This procedure adds a new user to the VM and pushes the user's SSH public key in the VM itself, so that you will be able to login using your private SSH keys. In addition, it also adds a password for the default user (`ubuntu`), allowing that user to log-in only through the console. This procedure may be convenient in case, for any reason, you loose the connectivity with your server, hence logging in via console is the only way to recover your VM. In any case, be careful that the access via username/password is considered less secure compared to the use of a public/private key.

1. If you do not have them already, create your pair of public/private keys with `ssh-keygen`. A good example can be found by following **Step 1** at `https://www.digitalocean.com/community/tutorials/how-to-set-up-ssh-keys-on-ubuntu-1604` (the following steps are not needed here).
2. Once done, create a new text file named `user-data`. For who is not very familiar with console-mode text editors in Linux, we suggest `nano`, which allows a reasonable user-friendly interaction (compared to earlier software such as `vim`).
3. Write in the above file the text **exactly** such as in the following example, while the `ssh-rsa` section should be replaced with your public key(i.e., all the text in your public key file, usually stored in your home folder, i.e. `~/.ssh/id_rsa.pub`) in the `ssh_authorized_keys` section of the above file:

[2]For security reasons, the default user account on the above image, i.e., user 'ubuntu', has the login disabled from both console and SSH.

[3]Latest documentation for `cloud-init` is available at `https://cloudinit.readthedocs.io/en/latest/`.

```
#cloud-config
users:
  - default
  - name: ubuntu
    ssh_authorized_keys:
      - ssh-rsa AAAA...DSHEx netlab@cloud-client
    sudo: ALL=(ALL) NOPASSWD:ALL
    groups: sudo
    shell: /bin/bash
# Password for the 'default' user (i.e., 'ubuntu' in Ubuntu cloud images)
password: ubuntu
# Password does not have to be changed at first login
chpasswd: { expire: false }
# Uncomment this line if you want to use username/password when
#  logging in also from network (ssh)
#ssh_pwauth: true
```

This code snipped is attached to the PDF file as "snippets/cloud-init.txt"

Warning: the `user-data` file relies on proper indentation to be correctly parsed. Pay attention to NOT use tabulations and ensure a consistent number of spaces across the file.

4. Create a second text file, called `meta-data`, with the following content:

   ```
   local-hostname: test-vm1
   ```

5. Use the tool `cloud-localds` to create a configuration disk, which is called *seed* in `cloud-init` terminology. Type the following command:

   ```
   cloud-localds seed-init.iso user-data meta-data
   ```

   to produce a file called `seed-init.iso`, which simply contains the above two files with a standard name, whatever name were assigned to your original files, i.e.:

   ```
   root/
    |
    +----/user-data
    |
    +----/meta-data
   ```

6. Copy the produced ISO into your `~/kvm-iso` folder:

   ```
   cp seed-init.iso ~/kvm-iso/.
   ```

Cloud-init and persistent disks

This lab uses KVM as virtualization framework, which makes persistent any change to your OS image. For instance, once you use your `cloud-init` file to configure your users, the above changes are written on the OS image on disk and are kept for the future runs of the above image. This is different from what happens in other systems (e.g., OpenStack), in which the OS image is never modified.

This brings us to two observations.

1. A first difference is that in our case the `cloud-init` disk may be required only in the first execution of the OS image, since in the following executions the added user will already be present; hence, we may modify the hardware profile of the VM by removing the `cloud-init`

disk, without any effect. Instead, the `cloud-init` initialization disk must be always provided in other systems (e.g., OpenStack) as the OS image is always the original one.

2. A second difference is that, in our environment, replacing an already applied `cloud-init` image with another one, with a different configuration data, may not work. In fact, `cloud-init` may not be able to change the configuration of an already present user, while it should be able to add a new user with another key. As a consequence, if the student wants to change its login credentials and by creating another `cloud-init` disk, it would be safer to re-initialize the OS image (e.g., replace the current OS image with the original Ubuntu cloud image) in order to be sure that `cloud-init` will work again.

5.5.2. Virt-customize

IMPORTANT: this section is left for your reference, but IT MUST NOT BE CARRIED OUT IN THE LAB.

`Virt-customize`[4] is an optional `libvirt` package[5] that can modify the original disk image. For instance, while `cloud-init` enables the VM to boot with an unmodified disk and changes the configuration of the operating system at boot time (by applying the desired configuration directives read from a local disk or from the network), `virt-customize` changes the original disk image by updating its file system.

For instance, the command to set the root password as '`coolpwd`' in an existing image named `ubuntu-18.04-minimal-cloudimg-amd64.img` is the following:

```
virt-customize -a ubuntu-18.04-minimal-cloudimg-amd64.img \
    --root-password password:coolpwd
```

Now you can login from console using username `root` and the password you set.

Note (1): the content of the VM image will be permanently changed.

Note (2): although `virt-customize` can also be used to add a new user (e.g., `netlab`), this command is more complicated because it requires some scripting. Since this lab suggests the use of `cloud-init`, more advanced customization actions with `virt-customize` are left to the good will of the student.

5.6. Create a new VM

In this step we operate on the **management station**, exploiting the GUI-based Virtual Machine Manager (`virt-manager`) that we presented in Section 5.2.1.

1. Add a new connection to the server, either from the graphical interface (`file->add connection`, then set the username and IP of the server), or leveraging the following command:

[4]Detailed documentation at `http://libguestfs.org/virt-customize.1.html`.
[5]In Ubuntu, this can be installed through `apt install libguestfs-tools`.

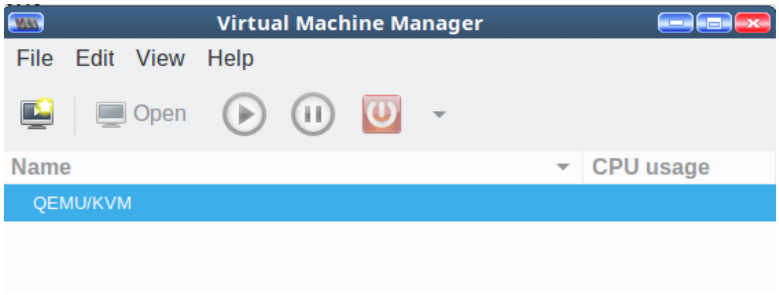

Figure 5.3: Virtual machine manager: main window.

```
virt-manager -c 'qemu+ssh://netlab@<SERVER_IP>/system?keyfile=<PRIVATE_KEY>'
e.g.
virt-manager -c 'qemu+ssh://netlab@10.100.101.102/system?keyfile=/home/netlab/.
    ssh/id_rsa'
```

This code snipped is attached to the PDF file as "`snippets/virt-manager.txt`"

Note (1): in case you use `virt-manager` to connect to a remote hypervisor through SSH, it is strongly encouraged to configure both server and client with SSH keys, as libvirt needs to open multiple SSH connections, one for each device (e.g., video card, keyboard, audio card, etc) that has to exchange data with the remote machine. Hence, relying on classical `user/password` mechanism would be rather annoying as the user would be requested to type his password several times.

Note (2): in case you prefer to connect to the remote hypervisor through username/password, you should install the `ssh-askpass` package on your client machine (`sudo apt install ssh-askpass`), so that `virt-manager` can open a new window for you to type the password in it.

Note (3): in case you are connecting from remote, using the ssh configuration file suggested in the lab slides, the above command line is ignored, as everything (except for the destination IP address) is read from the configuration file. Hence, you can simply use the traditional connection wizard (File - Add connection) and fill in the required data in there.

2. **Create a new VM** by going to *File >New Virtual Machine* (Figure 5.4). You can either:

   - Select **Import existing disk image**, then click Forward: select this option if you want to create a new VM by using a pre-installed disk. Note that the preinstalled image should be in one of the formats supported by KVM (e.g., QCOW/QCOW2, Vmware VMDK).

     Note: this is the preferred choice in this lab, as we have already a pre-installed disk, namely the Ubuntu Cloud Image retrieved in Section 5.4.

   - Select **Local Install Media** (ISO image or CDROM), then click Forward: select this option if you want to install a new VM using the installation disk. Note that the install disk should be in ISO format.

   This opens a window such as in Figure 5.5. In either cases, when you try to select the location of your image (either an ISO or a pre-installed image) by clicking on *Browse*, a new window *Choose Storage Volume* will appear (Figure 5.6), which contains only the `Default` storage pool, i.e., `/var/lib/libvirt/images`. Here you should configure the additional folders you created before:

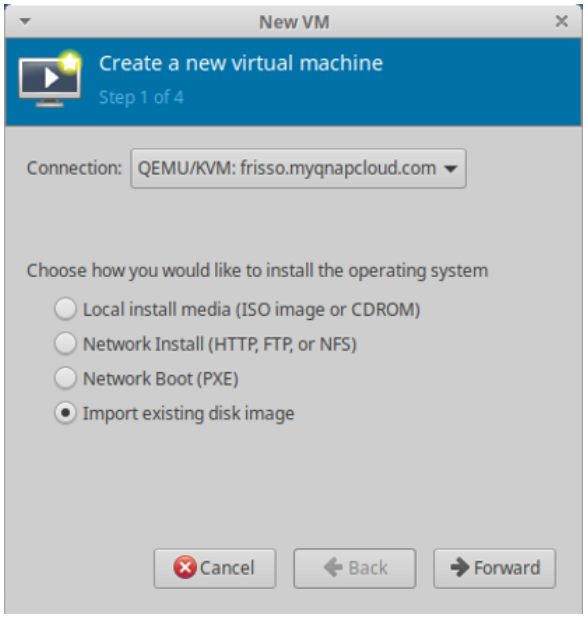

Figure 5.4: Create a new VM: step 1.

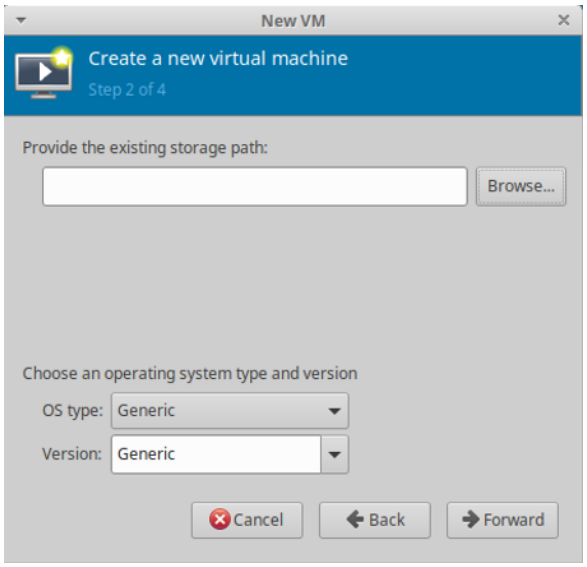

Figure 5.5: Create a new VM: step 2 (before choosing storage path).

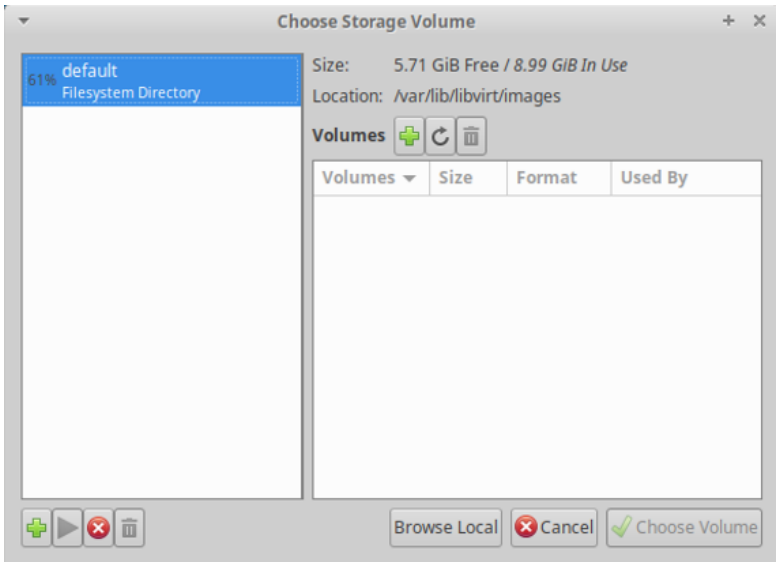

Figure 5.6: Available storage volumes (before adding the new storage pools)

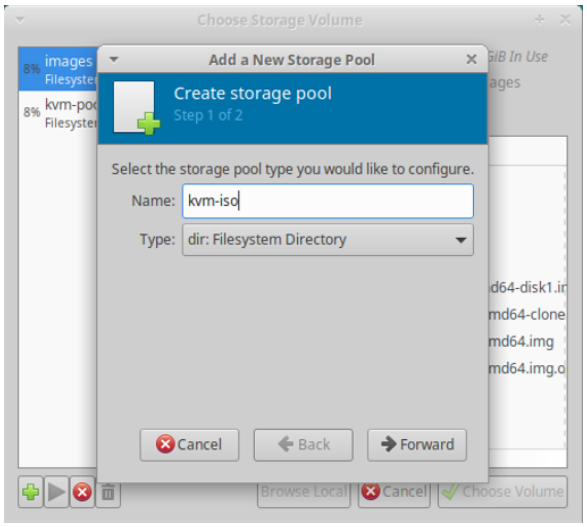

Figure 5.7: Create a storage pool: step 1.

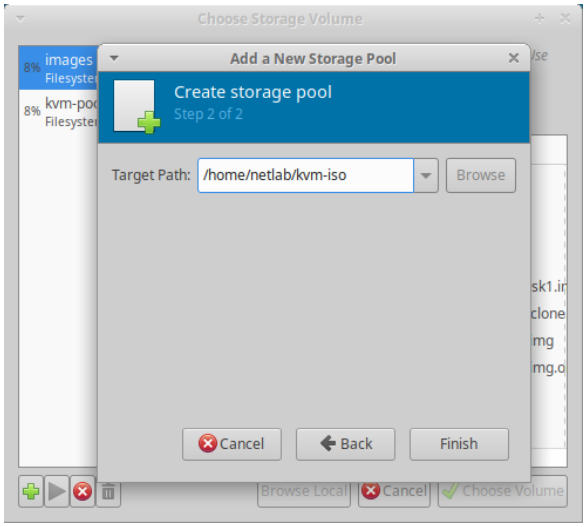

Figure 5.8: Create a storage pool: step 2.

- Select the '+' icon (**bottom right** in the previous window), to add a new storage pool.
- Choose a name for the new storage pool (e.g., `kvm-pool`), with type "`dir: Filesystem Directory`", as in Figure 5.7, and press the *Forward* button.
- Select the folder that has to be used for this storage pool (e.g., `~/kvm-pool`), such as in Figure 5.8 and press *Finish*. Note: if you are connected to a remote machine, the *Browse* button may not work; in this case you have to type the desired *Target Path* manually.
- Repeat the above steps to add also the `~/kvm-iso` folder as another storage pool.

Alternatively, you can simply click the *Browse Local* button at the bottom of the right panel to find the ISO/disk image you want to use, without adding new storage pools. In that case, just select your desired image and click *Open*.

3. Now that you are back at the *Choose Storage Volume* screen, you should see the default and your new storage pool in the left pane, with all the disk images/ISO that are available in that folder[6] (Figure 5.9). Select the disk image/ISO you want and click *Choose Volume*, you will return to the a window such as in Figure 5.10.

   Here, select the type of the OS you are installing (*Linux*), the version (*Ubuntu 18.04*) and select *Forward*.

4. Select the memory and the CPU to allocate to the above VM; given the constrained environment we are on, 512MB RAM and 1 CPU looks appropriate (Figure 5.11).
5. Once you reach last step (Figure 5.12), select "*Customize configuration before install*" in order to modify the virtual hardware of your VM and make the following modifications to the suggested profile:

   - Remove all unnecessary hardware (e.g., Display Spice, etc), leaving the minimal definition as shown in Figure 5.12;
   - Add a new **disk device** that is used to load the `cloud-init` configuration we created in Section 5.5.1.

[6] In case no disk images/ISO appear, refresh the window by clicking on the circular arrow, near to the text **Volumes**.

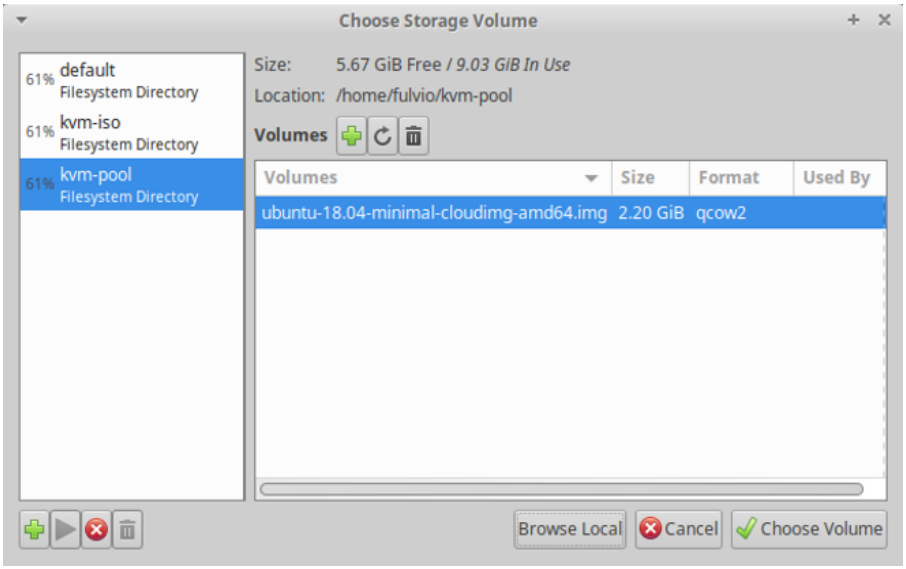

Figure 5.9: Available storage volumes (after adding the new storage pools).

Detailed steps: Select *Add Hardware* button, then *Add a new storage device*, then select `Disk device` as device type, then "*Select or create custom image*" and choose the ISO you created in Section 5.5.1).

Note: please note that the second disk should be a *disk device*; *CD-ROM devices* do not work.

- Expand the "*Network Selection*" section;
- Make sure the VM is connected to the "default" virtual network, with NAT. This enables your VM to connect to the Internet using the IP address of the hypervisor, while having a bridged connectivity to any other VMs connected to the same bridge.

You are now ready to start the VM and log-in.

Start and customize the VM

You can now start the VM and login in the new machine, with the credentials you set before.

Now we have to customize our environment, installing the `iperf3` tool, which will be used to generate some traffic and test the performance among VMs.

First, enter in superuser mode and reduce the MTU (otherwise the network traffic does not come to you):

```
sudo su
ip link set mtu 1400 dev enp1s0
exit
```

Now, we can download the `iputils-ping` package, which contains `ping`, and the `iperf3` package, which contains a minimal software for measuring network performance:

```
sudo apt update
sudo apt install iputils-ping iperf3
```

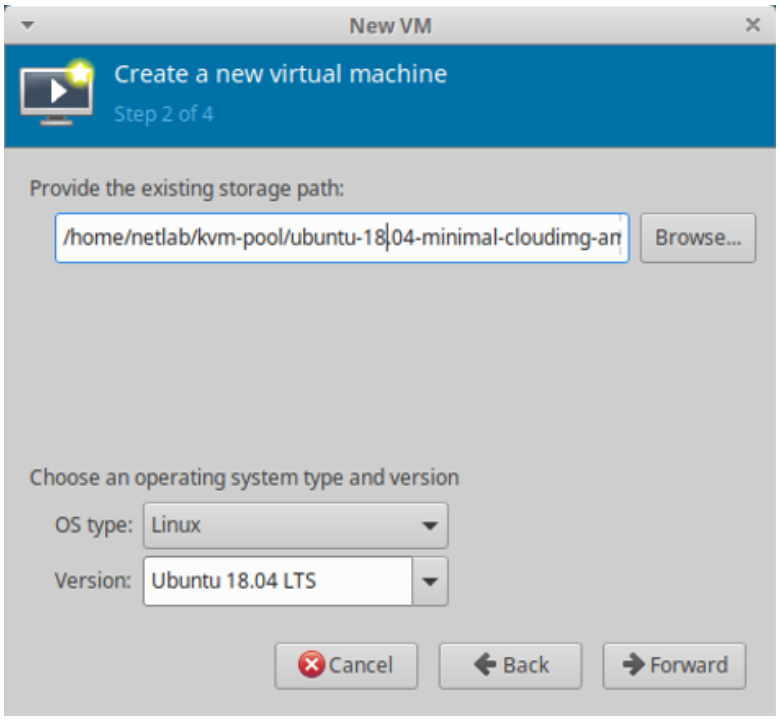

Figure 5.10: Create a new VM: step 2 (after choosing storage path).

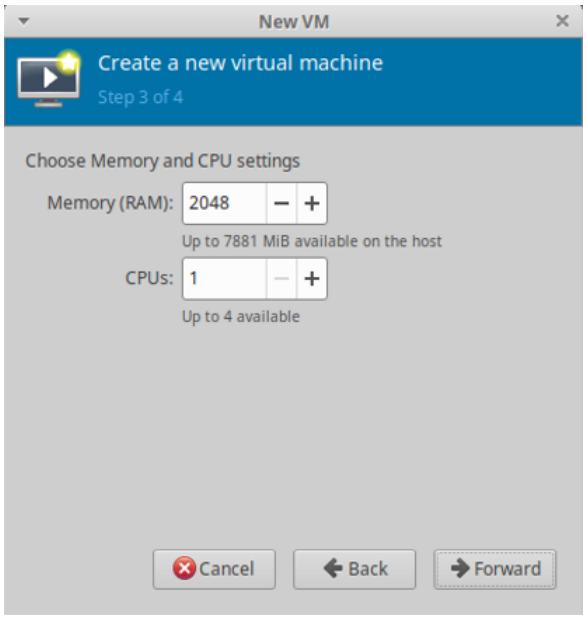

Figure 5.11: Create a new VM: step 3.

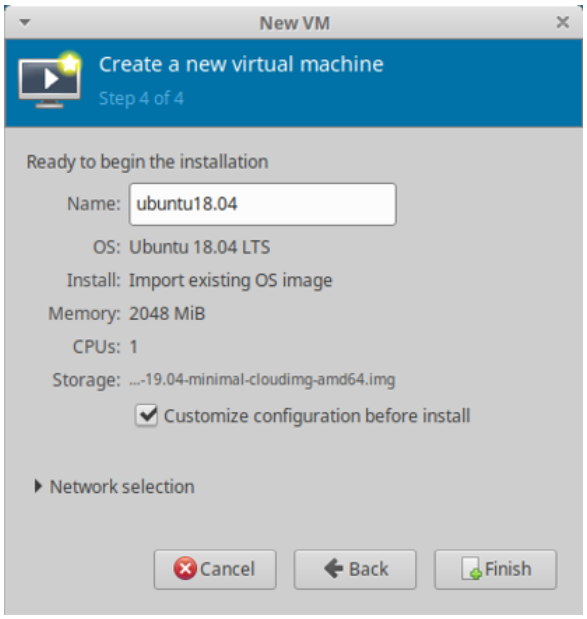

Figure 5.12: Create a new VM: step 4.

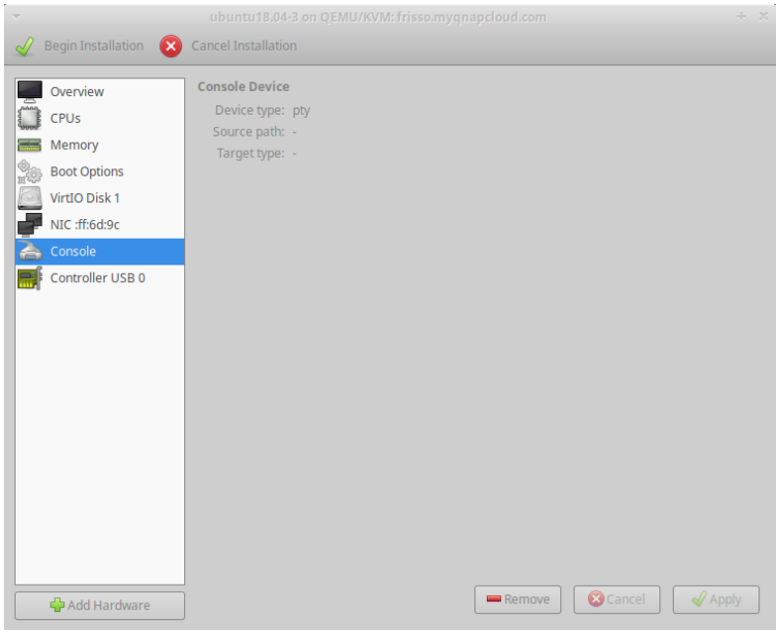

Figure 5.13: Virtual machine manager: minimal hardware.

Before cloning the vm, it is preferable to set up a password for the ubuntu user. With cloned vms, cloud-init detects that network configuration have been already generated and does not enable the default NIC interface and may require explicit commands to be set up. The password for user Ubuntu can be reset via:

```
sudo passwd ubuntu
```

Now we can close shutdown the VM, in order to clone it:

```
sudo shutdown -h now
```

5.6.1. Clone the existing VM into VM2

From the main window ov virt-manager, clone the VM. Follow the steps already shown before in order to create another initialization disk for `cloud-init`, in particular in order to show the name `test-vm2` as a prompt message.

Note: in case the second VM fails to bring up the network interface, use the following commands:

```
ip link set dev enp1s0 up
sudo dhclient enp1s0
```

Now, look at the IP address assigned (you can use the command `ip addr`) to each machine and make sure that the machines can ping each other.

5.6.2. Measure the network performance between the two VMs

To measure the network performance between the two VMs, you can the the following commands:

- On VM1, acting as server:

  ```
  iperf3 -s
  ```

- On VM2, acting as client:

  ```
  iperf3 -c <VM1_IP_address>
  ```

  where the IP address of VM1 can be detected by typing the command `ip addr` within VM1.

The `iperf3` tool will start a set of tests using TCP traffic and, at the end, it will return the average speed measured between the two machines.

5.7. Network Configuration

In this section we create a new virtual network, using the XML definition and manipulating it to create the network via the CLI. This is mainly done because manipulating the XML allow the possibility to have complete control of virtual hardware specification, by setting up also configuration which are not possible using the GUI (e.g. set up openvswitch as bridging technology).

As mentioned before, a subset of network configuration can be also set via the virt-manager GUI. We can create/delete/list virtual network of each hypervisor by selecting a connection, right-clicking on it and then open "Connection details". In this view, the virtual networks tab is available among "Overview" and "Storage".

Note: Unless when explicitly mentioned, all the following instructions are intended to be executed on the server.

5.7.1. Inspecting default network

After installing Libvirt on a Linux system,a "default virtual network" is provided out of the box. You can verify its configuration by typing:

```
# virsh net-list --all
Name                 State      Autostart
----------------------------------------
default              active     yes
```

This network is used now by the VMs just created in previous steps. To inspect more about the network configuration, it is possible to export every resource described in libvirt as as an XML file. The virsh CLI can help inspecting the configuration by "dumping" the xml of the network by typing:

```
sudo virsh net-dumpxml default
```

```
<network connections='2'>
  <name>default</name>
  <uuid>fe7073de-9253-4525-9529-9f0f9db345b9</uuid>
  <forward mode='nat'>
```

```
    <nat>
      <port start='1024' end='65535'/>
    </nat>
  </forward>
  <bridge name='virbr0' stp='on' delay='0'/>
  <mac address='52:54:00:82:cd:11'/>
  <ip address='192.168.122.1' netmask='255.255.255.0'>
    <dhcp>
      <range start='192.168.122.2' end='192.168.122.254'/>
    </dhcp>
  </ip>
</network>
```

The important parameters we can observe are the following:

- The **forward mode** element describes how the traffic has to be routed outside the VMs to the internet. In particular
  - **"nat"** forwarding (aka "virtual networks"): Like the "default" network, a virtual network switch operates in NAT mode (using IP masquerading). This means any guests connected through it, use the hypervisor IP address to communicate with outside destinations. This configuration does not allow incoming traffic to be received by this vNIC from the outside world.
  - **"routed"** networking (aka "shared physical device"): the virtual switch is connected to the physical host LAN, passing guest network traffic back and forth without using NAT. A "bridged" vNIC can configure an IP in the network where the NIC of the hypervisor is bridged, and be contacted from the outside world.
  - If the forward section is absent, the network is isolated from outside.
- **bridge name** element indicates which software bridge has to be used to connect virtual NICs of VMs belonging to this network.
- The **IP** element: each virtual network switch can be given a range of IP addresses, to be provided to guests through DHCP. Libvirt uses a program, dnsmasq, for this. An instance of dnsmasq is automatically configured and started by libvirt for each virtual network switch needing it.

When added to a network, vNICs are attached to the virtual bridge indicated in the network specification. To verify this, we can simply inspect the virbr0 bridge, to check on which vNICs are connected:

```
sudo brctl show

bridge name bridge id          STP enabled    interfaces
virbr0      8000.52540082cd11     yes         virbr0-nic
                                              vnet0
                                              vnet1
```

In addition we can find which MAC addresses corresponds to which vNIC and detect which VM is actually attached:

```
sudo brctl showmacs virbr0
```

From virt-manager, clicking on a specific VM on the hardware view, we may obtain the NIC MAC address.

5.7.2. Create a new virtual network

Starting from the default network, extract the network configuration by dumping it into a file:

```
sudo virsh net-dumpxml default > br1.xml
```

Now, in order to create a new network modify the following fields in br1.xml:

- Change the network name, two different networks cannot have the same name.
- Delete the network UUID. It will be regenerated by the Libvirt daemon at creation time.
- Change the bridge name. If the bridge does not exists, it will created by the Libvirt daemon.
- To create an internal network, remove the forwarding section.
- Assign another subnet in the IP element.

Check if your network file is correct by typing:

```
sudo virsh net-define br1.xml
```

The, start the new network:

```
sudo virsh net-start br1.xml
```

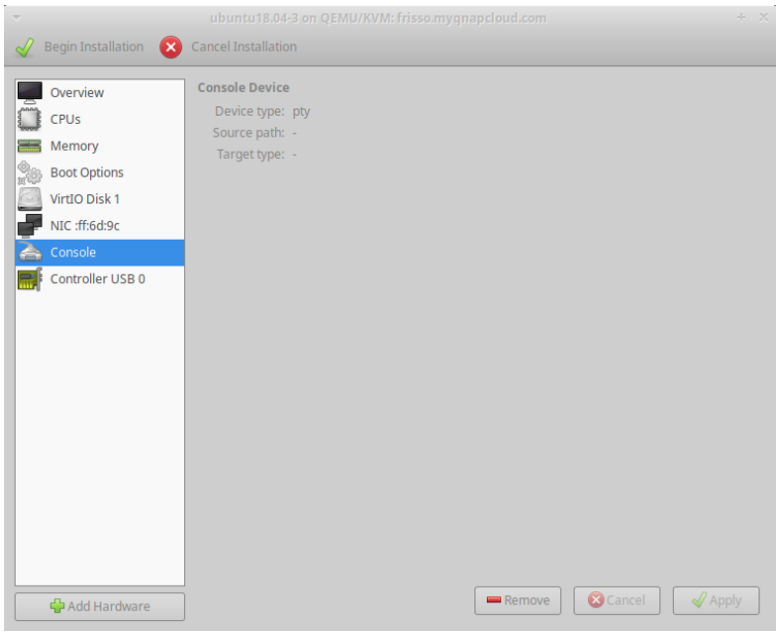

Figure 5.14: Virtual machine manager: add a new NIC from the hardware info section

If the network is created with success, we can create a new NIC in the two VMs created before (you will need to shutoff those VMs to apply this modification). New NICs can be created by the hardware information window as showed in figure 5.14 and attach the new NIC to the newly created network We can test connectivity using iperf in this network as we did with the default network.

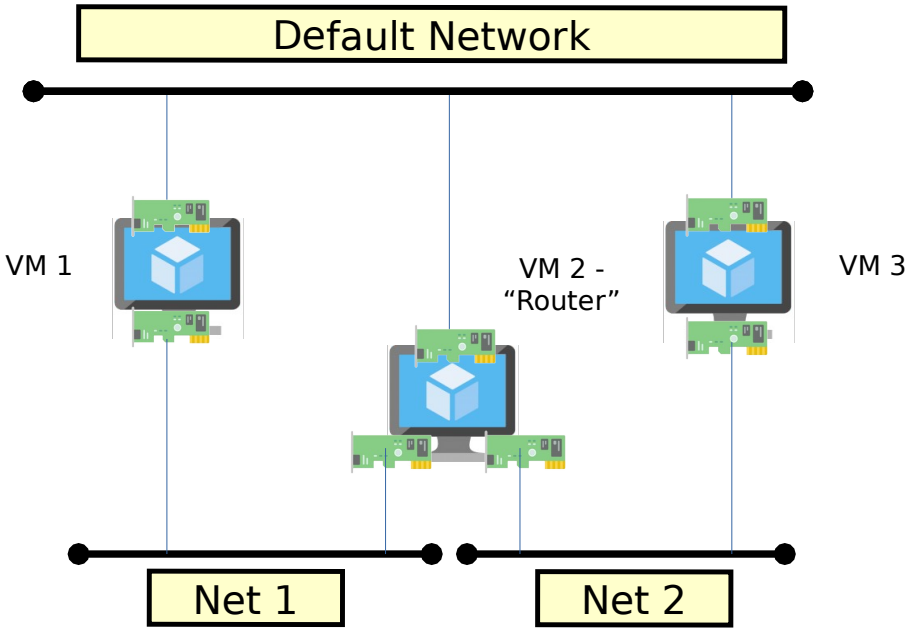

Figure 5.15: VM1 communicates with VM3 via VM2, used as a router. All VMs have a management NIC to support ssh connection from the outside world.

5.8. Use a router VM to exchange traffic between two VMs

In this section, it is proposed a more advanced exercise combining topics discussed in the previous sections. Therefore, instructions are less extensive and it is up to the reader combine them together to satisfy the requirements.

As we discussed in previous section, virtual networks can be combined to create multiple topologies and route traffic across VMs. In this exercise, we create the topology depicted in figure 5.15: two VMs should communicate by using a dedicated secondary vNIC connected via a ”router” VM.

- Create 2 internal networks without overlapping IPs.
  Suggestion (1): We suggest to enable DHCP in order to simplify IP assignation.
- Create two VMs with two NICs each with the **VirtIO driver**. The first NIC should be connected to the ”default” virtual network or another virtual network accessible from the outside. The second one should be connected to one the two virtual networks respectively.
- Create the ”Router” VM and attach three vNICs to it, one for the management network and the other two to be interconnected to the internal virtual networks.
- Configure NICs in the 3 VMs in order to enable routing. In particular, we should enable `ip_forwarding` on the ”router” VMs and the appropriate routes on the other ones.
- Several tests that can be performed to check that the topology is implemented:
  - Ping the IP of vNIC of VM3 to assess reacheability of the VM.
  - Use traceroute `IP_VM3` to verify that your packets correctly traverse the router.
  - Test bandwidth using iperf as presented in 5.6.2 between VM1 and VM3.

The results obtained by testing the performance of iperf in this exercise will be used in the next one to compare the performance of paravirtualized and emulated network card.

5.8.1. Useful Commands

To complete the previous exercise, the following commands should complement what we observed on the

- Enable "IP forwarding" to let a Linux system act as a router:

```
echo 1 > /proc/sys/net/ipv4/ip_forward
```

- Create a dedicate route using `ip route`

```
ip route <destination_network> via <gateway_ip>
```

- Request an IP using dhcp:

```
dhclient -v <interface_name>
```

5.9. Extra Exercises

This section provides some further and more advanced exercises that can be useful to enlarge skills using cloud technologies.

5.9.1. Compare performance of different network devices

To complete this exercise, we consider the same topology adopted in the previous exercise, and depicted in 5.15 By default, virtual machines are created with NICs and hard-drives using the VirtIO driver. VirtIO driver implements a paravirtualized device which is promised to be faster then traditional emulated device. In this exercise, we experimentally evaluate the throughput obtained with an iperf test with vNICs using VirtIO and emulated NICs (e1000).

To perform this test, we change the technology used by all vNICs involved in the routing "path", from **VirtIO to e1000**. This can be simply done by (1) editing the XML and changing manually the definition of a VM and (2) via virt-manager by entering in the vm details -¿ list hardware devices and then for every NIC, change the "Device Model" to **e100/e100e**.

The result show that VirtIO paravirtualized devices performs consistently better than emulated one and this justify their adoption by default when VMs are created.

5.9.2. Create the 5.8 using a "Infrastructure as Code" approach

Vagrant (`https://www.vagrantup.com/intro` is a tool for building and managing virtual machine environments. Vagrant is particularly designed to automate deployment and creation of VMs for continous integration. Virtual Machines and their configuration can be described using Vagrantfiles and launched using simple "vagrant up". In this exercise, let's try to reproduce automatically the same topology by leveraging Vagrant. The objective is to be able with a single command (e.g.; vagrant up), to deploy the three VMs of the exercise and their network configuration,

Part IV.

Appendix

6. KVM setup

If you are using the VM provided with the lab, you can skip this section. In fact, the VM already contains all the tools needed for the lab assignment. This section will be useful if you have to rebuild a VM with the required environment from scratch.

KVM is a Linux kernel module that turns the standard Linux operating system into an hypervisor, hence introducing kernel support for VMs.

In order to install KVM on a recent Ubuntu distribution, follow the detailed instructions at `http://help.ubuntu.com/community/KVM/Installation`. We strongly suggest to install also the `virt-manager` GUI, which is marked as optional in the above guide.