TÉCNICO LISBOA

RC 20/21	LAB INTRODUCTION	Guide #:	0
Lab Environment Preparation		Issue Date:	Sept 2020
The Mininet Network Emulator		Due Date:	
Author: Prof. Rui Cruz		Revision:	4.3

Preliminary Notes

One nice feature of the software stack we are going to use is that it is portable to many platforms including **YOUR OWN** personal computers. You will find the guidelines to configure your laptop or desktop properly at the Section 1. The instructions are suitable for machines running the following Operating Systems:

- Microsoft Windows from version 10 up
- Apple macOS from versions 10.13 'High Sierra' up
- Debian-based Linux, such as Ubuntu (recommended) from versions 12.04 'Precise' up.

It is **not recommended** to apply this setup to a virtual machine (**nested virtualization**), although possible, as the configuration requires access to a hypervisor environment (mandatory Virtualbox) in the host system.

For the aforementioned Operating Systems, the setup will use 'package managers' in order to turn the process well controlled (versions, updates), simple to 'cleanup' and not 'affecting' the host environment.

Note: Avoid copying text strings from the command line examples or configurations in this document, as pasting them into your system or files may introduce/modify some characters, leading to errors or inadequate results.

1 Setting up the Environment

This section describes the minimum setup of the experimental environment for systems running Windows, macOS and Linux.

We will use the following tools or programs:

wget : a non-interactive tool to download files from the web, using HTTP/HTTPS.

git : is a distributed version-control system for tracking changes in source code.

virtualbox : Oracle VM VirtualBox is a free and open-source hosted hypervisor for x86 virtualization.

vagrant : is an open-source product for building and maintaining portable virtual development environments.

vcxsrv : [WINDOWS 10] is a X-Windows (X11) server program that provides graphical (GUI) interface in the Windows (host) environment from applications running inside Unix/Linux systems.

xquartz : [macOS] is a X-Windows (X11) server program that provides graphical (GUI) interface in the macOS (host) environment from applications running inside Unix/Linux systems.

atom : Atom is a free and open-source text and source code editor.

Vscode : Visual Studio Code is a free and powerful, based on open-source, code editor from Microsoft.

1.1 Microsoft Windows Setup

The recommended **Package Manager** for Microsoft Windows is Chocolatey `https://chocolatey.org/`, similar to Linux *apt* or *yum*. Chocolatey was designed to be a decentralized framework for quickly installing Windows applications and tools, and is built on the NuGet infrastructure, currently using Windows PowerShell.

On your machine, it is recommended that you create a directory (folder) for your projects, named for example MyProjects.

IMPORTANT: you MUST **ensure that there are no spaces in the PATH name**.

```
C:\>  mkdir C:\MyProjects
```

To install Chocolatey, open a privileged (i.e. administrative) Windows Command Prompt (cmd.exe) and paste the text string exemplified (at the command prompt) and press Enter. Please refer to the Chocolatey website for the available install options: `https://chocolatey.org/install`. You should have a string similar to the following (do not use this one, it is just an example):

```
C:\> @"%SystemRoot%\System32\WindowsPowerShell\v1.0\powershell.exe
   " -NoProfile -InputFormat None -ExecutionPolicy Bypass -Command
    "iex ((New-Object System.Net.WebClient).DownloadString('
   https://chocolatey.org/install.ps1'))" && SET "PATH=%PATH%;
   %ALLUSERSPROFILE%\chocolatey\bin"
```

After Chocolatey installed, it is now time to install the other packages necessary for running the Lab experiments. Install the following programs using Chocolatey. Please note that for GIT, the optional parameter also installs “Unix Tools” for Windows, such as “git-bash” that provides a command-line Terminal, and an SSH client:

```
C:\> choco install wget
C:\> choco install git.install --params "/GitAndUnixToolsOnPath"
C:\> choco install virtualbox
C:\> choco install vagrant
C:\> choco install vcxsrv
C:\> choco install atom
```

After this point, and in order to settle everything (Windows is quite tricky in updating environment variables), close the Console. Additionally, as an SSH (secure shell) client was not distributed with Windows until recently, it may be necessary to tell the system where to find the SSH client installed with GIT:

- Open the Control Panel
- Go to System and Security
- Click on System, then on the Change Settings button
- Display the Advanced tab and click on Environment Variables...
- Look for the Path variable in the System variables list, select it then Edit...

At the end of the string, confirm if exists a path such as the following, otherwise add the path pointing to Git's bin folder:

```
C:\Program Files\Git\bin\;C:\Program Files\Git\usr\bin
```

For the installed programs to work properly in Windows, some environment variables need to be set. You can now open a Terminal window using Git-Bash (that you should "pin to the taskbar" and configure its proprties to "run as Administrator"). The commands that you will use (as exemplified next) will create a "user resources" script `.bashrc` and then launch the Atom editor. :

```
user@computer MINGW64 /
$ cd

user@computer MINGW64 ~
$ touch .bashrc

user@computer MINGW64 ~
$ atom .bashrc
```

In the editor window insert the string, as exemplified, for a **System Variable** with name **DISPLAY** and with **Variable value** of `localhost:0.0`, then save.

You will need to close and reopen the Git-Bash Terminal in order for the "user" resources to be in effect.. For the XWindows server, start **VcXsrv** (there is an icon named XLaunch) as illustrated in Figure 2, accept the "default" configurations, and then an "X" icon of the server will be visible in the Taskbar system tray. Hovering the mouse over it should display something like `computername:0:0 - 0 clients`.

1.2 Apple macOS setup

The recommended Package Manager for macOS is **Homebrew**, similar the Linux *apt* or *yum* `http://brew.sh`. Homebrew installs packages (binary or to be compiled) to

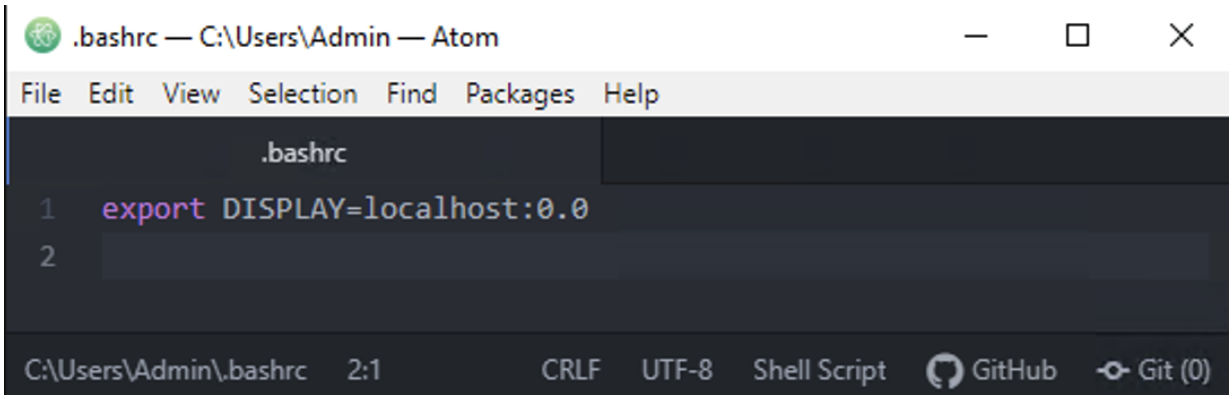

Figure 1: Setting New System Properties DISPLAY Variable

their own directory and then symlinks their executables into */usr/local*. Homebrew provides Homebrew-Cask, implemented as a Homebrew external command called *cask*. Homebrew-Cask extends Homebrew and brings its elegance, simplicity, and speed to the installation and management of GUI macOS applications such as Google Chrome.

As a first step you may need to install Command Line Tools for **XCode** from Apple (free from the App Store), in order for Homebrew to be able to compile applications from their source code. Macs do not have any of the developer's 'Command Line tools' installed by default, so we need to install them before we can get anywhere. If you do not have **XCode** installed then open Terminal and at the shell prompt type the following:

```
:~$ xcode-select --install
```

To install Homebrew, at the shell prompt copy the install script that you can find at `http://brew.sh`, similar to the following:

```
:~$ /usr/bin/ruby -e "$(curl -fsSL https://raw.githubusercontent.
   com/Homebrew/install/master/install)"
```

The shell prompt will tell you what it is about to do, and ask you if you want to proceed: press **Enter** to do so. The shell prompt may then ask for a password: this is the password to the Admin account on your computer. Type your password, and press **Enter**. When it's done, the shell prompt will say that the installation was successful, and ask you to run **brew doctor**. Do as it suggests:

```
u:~$  brew doctor
```

This will make Homebrew inspect your system and make sure that everything is set up correctly. If the shell prompt informs of any issues, you will need to fix them, and then run **brew doctor** again to verify that all was correctly fixed. When everything is set up correctly, you will see the message '*Your system is ready to brew*', and so, move on.

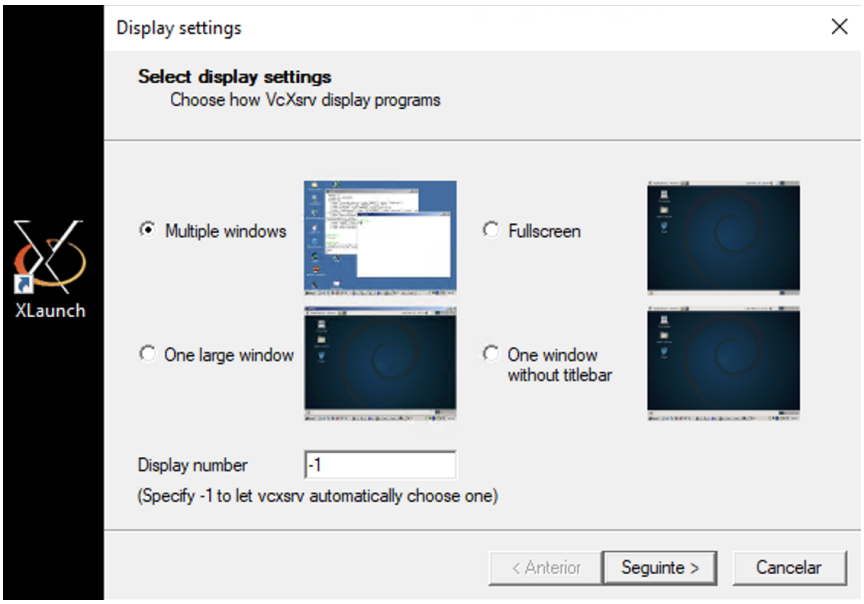

Figure 2: Starting X-Windows server

It is now time to install the other packages for the Lab experiments environment. Open Terminal and install the following programs using Homebrew:

```
:~$  brew install wget
:~$  brew install git
:~$  brew cask install virtualbox
:~$  brew cask install vagrant
:~$  brew cask install xquartz
:~$  brew cask install atom
```

For each command the Terminal will start displaying lots of information, as Homebrew is keeping you updated on what it is doing. This flow of information will be a guide to let you know whether or not the computer is still working, and so, do not interrupt it.

Now it is time to create a directory/folder for your Lab projects, for example:

```
:~$  mkdir ~/Desktop/MyProjects
```

1.3 Linux

For a Linux system, namely Ubuntu, the standard package manager 'apt-get' is already present, as well as some packages, such as 'wget'. To install the other packages open a Terminal and issue the following commands (answer Y to the prompts, and several dependencies will also be installed).

```
:~$  sudo apt-get install git
```

Please note that for latest distributions of Ubuntu, a careful install of virtualbox is required as the package architecture has to match the Linux kernel architecture. Additionally, the VirtualBox package in Ubuntu may have some issues when running in RAM-constrained environments. For that purpose you need to ensure that you have the adequate sources referenced for your kernel architecture. The following examples are for common Ubuntu versions, such as Ubuntu Xenial (16.04). More recent versions may require different procedures.

Start by editing the sources list issuing the following command:

```
:~$  sudo nano /etc/apt/sources.list
```

Add the following line to the list (exemplified for 16.04 'Xenial' kernel):

```
deb http://download.virtualbox.org/virtualbox/debian xenial
   contrib
```

According to your distribution, replace 'xenial' by 'vivid', 'utopic', 'trusty', etc.

You also need to add and register the Oracle public key (combined command):

```
:~$ wget -q https://www.virtualbox.org/download/oracle_vbox_2016.
   asc -O- | sudo apt-key add -
:~$ wget -q https://www.virtualbox.org/download/oracle_vbox.asc -O
   - | sudo apt-key add -
```

Now you can add Virtualbox (be careful as the version for your system may be different from the one exemplified):

```
:~$  sudo apt-get update
:~$  sudo apt-get install virtualbox-6.0
```

For Vagrant there were issues and we need to get the latest version (perhaps not in Ubuntu repository) by downloading the package from the Vagrant website `http://downloads.vagrantup.com/` and then issuing command similar to the following (from the downloads folder, adapt to your version):

```
:~$ wget https://releases.hashicorp.com/vagrant/2.2.5/vagrant_2
   .2.5_x86_64.deb
:~$  dpkg -i vagrant_2.2.5_x86_64.deb
```

For a Linux system, namely Ubuntu, the XWindows system is normally already installed. You may just need to install the Atom editor, or other code editor you prefer, if needed.

```
:~$  sudo apt-get update
:~$  curl -sL https://packagecloud.io/AtomEditor/atom/gpgkey |
   sudo apt-key add -
```

```
:~$ sudo sh -c 'echo "deb [arch=amd64] https://packagecloud.io/
   AtomEditor/atom/any/ any main" > /etc/apt/sources.list.d/atom.
   list'
:~$  sudo apt-get update
:~$  sudo apt-get install atom
```

Reboot the system and create a directory for your Lab projects, for example:

```
:~$  mkdir ~/Desktop/MyProjects
```

2 Vagrant Environments

If you have ever tried to create Virtual Machines used for testing through a Graphical User Interface (GUI), you know that it can be a pain, and it is a very manual process (installing the Operating System, the packages/applications, configure them, etc.). There is also a tendency to leave testing machines around in your computer (even running and consuming precious resources) for a long time without rebuilding them (or even shutting them down). Before using Vagrant there was a natural resistance to create clean environments, because there is an extra labour cost associated with making this happen, as it is just a very manual process via a GUI.

Vagrant eliminates much of that extra labour, as most of it is completely automated so let'clear s dive a little deeper in Vagrant.

If you run the `vagrant` command without any arguments, you will get the default help output, which displays available command options:

```
u:~$  vagrant
Usage: vagrant [options] <command> [<args>]

    -v, --version                    Print the version and exit.
    -h, --help                       Print this help.

Common commands:
     box             manages boxes: installation, removal, etc.
     connect         connect to a remotely shared Vagrant environment
     destroy         stops and deletes all traces of the vagrant machine
     global-status   outputs status Vagrant environments for this user
     halt            stops the vagrant machine
     help            shows the help for a subcommand
     init            initializes a new Vagrant environment by creating a Vagrantfile
     login           log in to HashiCorp's Atlas
     package         packages a running vagrant environment into a box
     plugin          manages plugins: install, uninstall, update, etc.
     port            displays information about guest port mappings
     powershell      connects to machine via powershell remoting
     provision       provisions the vagrant machine
     push            deploys code in this environment to a configured destination
     rdp             connects to machine via RDP
     reload          restarts vagrant machine, loads new Vagrantfile configuration
     resume          resume a suspended vagrant machine
     share           share your Vagrant environment with anyone in the world
```

```
    snapshot        manages snapshots: saving, restoring, etc.
    ssh             connects to machine via SSH
    ssh-config      outputs OpenSSH valid configuration to connect to the machine
    status          outputs status of the vagrant machine
    suspend         suspends the machine
    up              starts and provisions the vagrant environment
    vbguest
    version         prints current and latest Vagrant version

For help on any individual command run `vagrant COMMAND -h`

Additional subcommands are available, but are either more advanced
or not commonly used. To see all subcommands, run the command
`vagrant list-commands`.
```

The main command options of Vagrant are used for managing the life cycle of Vagrant *boxes*. Box, or boxes, is a Vagrant term for virtual machine "template" images. Vagrant boxes are "special virtual machines", in that they typically have several required packages already installed, and a **vagrant user** added inside the image. The `vagrant box` option allows to list, add, and remove boxes that Vagrant knows about in your syustem:

```
u:~$  vagrant box
Usage: vagrant box <subcommand> [<args>]

Available subcommands:
     add
     list
     outdated
     remove
     repackage
     update

For help on any individual subcommand run `vagrant box <subcommand> -h`
```

The `vagrant init` option allows to initialize a new Vagrant environment (without specific customizations). A Vagrant environment can be a single Vagrant Virtual Machine (or Docker Container), or a collection of Virtual Machines (or Containers). So, an environment will describe what boxes, or Virtual Machines (and/or Containers to boot, along with all of the associated settings, through that configuration file named `Vagrantfile`.

The other common command options are `vagrant up` to boot the environment, `vagrant halt` to shutdown the environment, `vagrant global-status` to check the status of instantiated virtual machines, `vagrant destroy` to completely destroy each "instance" of a Virtual Machines, and `vagrant ssh` to establish a session with each virtual machine that is instanced and running.

In order to ensure adequate updating and upgrading for Vagrant environments using the Virtualbox provider add to Vagrant the `vagrant-vbguest` Plugin with the command:

```
:~$ vagrant plugin install vagrant-vbguest
# or, if already installed
:~$ vagrant plugin update vagrant-vbguest
```

3 Trial Experiment

The first experiment, valid for all the systems (Windows, Mac and Linux) will consist on the launch of an Emulated network using Mininet `http://mininet.org`. Mininet creates a realistic virtual network, running real kernel, switch and application code, on a single machine (in this case, a Virtual Machine). For that purpose it is necessary to provision the Mininet virtual machine using Vagrant.

First you need to position the creation of the system in the adequate working folder, named **mininet** under **MyProjects**.

For example:

- in Windows `C:\MyProjects\mininet`,
- in Mac or Linux `~/Desktop/MyProjects/mininet`

Then, from the course website in Fenix, download for this experiment the file named "**Vagrantfile**" to the mininet folder. This Vagrantfile is already customized for a smooth startup of the system.

The system needs to be started up with the command:

```
:~$ vagrant up
```

This phase may take up a few minutes, depending on the speed of the host system and your access to the Internet, with an output similar to the following:

```
Bringing machine 'mininet' up with 'virtualbox' provider...
==> mininet: Checking if box 'ktr/mininet' is up to date...
==> mininet: Clearing any previously set forwarded ports...
==> mininet: Clearing any previously set network interfaces...
==> mininet: Preparing network interfaces based on configuration
   ...
    mininet: Adapter 1: nat
==> mininet: Forwarding ports...
    mininet: 22 => 2222 (adapter 1)
==> mininet: Booting VM...
==> mininet: Waiting for machine to boot. This may take a few
   minutes...
    mininet: SSH address: 127.0.0.1:2222
    mininet: SSH username: vagrant
    mininet: SSH auth method: private key
==> mininet: Machine booted and ready!
==> mininet: Checking for guest additions in VM...
    mininet: Guest Additions Version: 4.3.10
    mininet: VirtualBox Version: 5.0
==> mininet: Mounting shared folders...
```

If you are installing from the box in the Internet (Vagrant Cloud repository) instead of our local network, it is possible that you get an error message such as the following:

```
Vagrant was unable to mount VirtualBox shared folders. This is
   usually because the filesystem "vboxsf" is not available. This
   filesystem is made available via the VirtualBox Guest Additions
    and kernel module. Please verify that these guest additions
   are properly installed in the guest. This is not a bug in
   Vagrant and is usually caused by a faulty Vagrant box.
```

This error message is a result of the outdated version of the Virtualbox drivers inside the box and you can safely ignore it.

Next, you will establish a session with the system using the following command:

```
:~$ vagrant ssh
```

The session is established and you will get the machine prompt similar to:

```
Welcome to Ubuntu 14.04.4 LTS (GNU/Linux 3.13.0-87-generic x86_64)
Last login: Sun Sep 20 22:50:28 2017 from 10.0.2.2

vagrant@mininet:~$
```

Once you get **ssh** access, update the packages to fix the problem with the old Virtualbox drivers using the following command (**skip this step** if you are using the **Mininet** box that you have downloaded):

```
vagrant@mininet:~$ sudo apt update && sudo apt upgrade -y
vagrant@mininet:~$ exit
# back in the local machine shell:
:~$ vagrant reload
# wait while the machine boots up....
:~$ vagrant ssh
```

A very simple test can then be performed to verify that all is working correctly by issuing the command 'sudo mn' at the Mininet shell prompt, to create a simple network topology of two hosts (h1, h2) and a switch (s1):

```
vagrant@mininet:~$ sudo mn
*** Creating network
*** Adding controller
*** Adding hosts:
h1 h2
*** Adding switches:
s1
*** Adding links:
(h1, s1) (h2, s1)
```

```
*** Configuring hosts
h1 h2
*** Starting controller
c0
*** Starting 1 switches
s1 ...
*** Starting CLI:
mininet>
```

From this point you can issue the command 'pingall' to verify if you have connectivity between the hosts:

```
mininet> pingall
*** Ping: testing ping reachability
h1 -> h2
h2 -> h1
*** Results: 0% dropped (2/2 received)
```

With the command 'net' you can verify how the nodes are connected, i.e., **host1** is connected by its **eth0** port to the **switch1** on **eth1** port, etc.:

```
mininet> net
h1 h1-eth0:s1-eth1
h2 h2-eth0:s1-eth2
s1 lo:  s1-eth1:h1-eth0 s1-eth2:h2-eth0
c0
```

Issuing the command 'dump' you can see the IP addresses of the hosts (10.0.0.1 and 10.0.0.2):

```
mininet> dump
<Host h1: h1-eth0:10.0.0.1 pid=1708>
<Host h2: h2-eth0:10.0.0.2 pid=1712>
<OVSSwitch s1: lo:127.0.0.1,s1-eth1:None,s1-eth2:None pid=1717>
<Controller c0: 127.0.0.1:6633 pid=1701>
```

To end the Mininet emulation you use the command 'exit' and it is also advisable to 'clean' any phantom Mininet processes in the system with the command 'sudo mn':

```
mininet> exit
*** Stopping 1 controllers
c0
*** Stopping 2 links
..
*** Stopping 1 switches
s1
*** Stopping 2 hosts
```

```
h1 h2
*** Done
completed in 5030.655 seconds
vagrant@mininet:~$ sudo mn -c
*** Removing excess controllers/ofprotocols/ofdatapaths/pings/
   noxes
***
*** Cleanup complete.
vagrant@mininet:~$
```

You can now exit the session:

```
vagrant@mininet:~$  exit
logout
Connection to 127.0.0.1 closed.
```

In order to stop the Virtual Machine and to verify the global state of all active Vagrant environments on the system, you can issue the following commands:

```
:~$ vagrant halt
==> mininet: Attempting graceful shutdown of VM...
:~$ vagrant global-status
```

Confirm that the statuses of the VMs is 'powered off'.

In Lab computers, also do the following (answering "y"), in order to ensure the system is cleaned from your experiments. Note however that the configuration files remain stored in your private area in AFS:

```
:~$ vagrant destroy
mininet: Are you sure you want to destroy the 'mininet' VM? [y/N]
==> mininet: Destroying VM and associated drives...
:~$ vagrant global-status
```

Confirm that there are no VMs listed.