UNIVERSITY OF
TECHNOLOGY SYDNEY

49202: Communication Protocols
Virtual Laboratory Overview & Instructions

Dr. Daniel R. Franklin

February 26, 2021

Contents

1 Introduction

The laboratory for this subject uses a software platform called Mininet, which uses a combination of Linux namespaces and virtual switches to emulate a network with arbitrary topology and link characteristics. Because it is based on Linux, and the emulated hosts and routers share most of their filesystem with the base operating system, you can run any server, service, client, p2p application etc. on them by simply installing it on the base Linux system and starting it in the context of the emulated host or router. This includes things like routing protocols, web servers and browsers, DNS and SSH servers. Anything that runs on Linux can (theoretically) run on one of these emulated hosts or routers.

We provide a virtual machine image that you may download and install on your own computer. You can get the latest version via the following link:

`https://drive.google.com/open?id=1AigxQ87oKPKGmdD6RTH28fRT1HezEzCn`

or from the Class Materials folder on Teams/SharePoint.

This image can be installed either in VirtualBox (a free, open source virtualisation platform), VMWare Player or KVM (for Linux users only) - it should work equally well in any of these if you already have one of them set up on your system. **The remainder of this laboratory assumes you are using VirtualBox**, although the procedure is similar on the other VM platforms. VirtualBox may be downloaded from

`https://www.virtualbox.org/wiki/Downloads`

After installing VirtualBox, download the VM image (a single 3.5 GB .ova file) from Google Drive or Teams/SharePoint, start VirtualBox and import the appliance (**File→Import Appliance**). At this stage you may modify the number of CPU cores (at least 2 is recommended - if you have a CPU with lots of physical cores, you may want to increase this) and the amount of memory (the default image will be allocated 4 GB of RAM, but if your laptop has 16 GB or more, you may want to increase this to 8 GB). Once the import process is complete, the downloaded .ova file may be deleted (or saved to a removable drive). You should now be able to start the VM in VirtualBox. You can save the state of the VM at any stage when you close the VM; you can alternatively do a full shutdown.

If your screen has a very different resolution to the default VM screen resolution, you can change the size of the virtual screen by going to **View→Screen** and choosing a different screen resolution - it should normally be zoomed to 100%.

Troubleshooting tip: some laptops have 'Intel Hardware Virtualisation' disabled in the BIOS. If you can't start the VM, you may need to go into your BIOS Setup (typically by rebooting and pressing a function key - you can search online for how to do this with your particular laptop) and enabling the option (sometimes called 'Intel VT-x'). A brief guide on how to do this is available at

`https://www.howtogeek.com/213795/how-to-enable-intel-vt-x-in-your-computers-bios-or-uefi-firmware/`

2 Maintenance

To keep the laboratory scripts up to date, firstly ensure that Internet connectivity is working (you can test from inside the VM by doing a `ping -c 5 google.com`). Then open a terminal window

and enter the `comm-protocols-labs` folder under your home directory (which is already set up on the VM image) and do a `git pull`:

```
cd ~/comm-protocols-labs

git pull
```

If there have been any updates to the Git repository since the VM image was built, this command will ensure that you have the latest available version installed.

3 Brief Introduction to Linux

This section summarises some basic Linux commands and concepts that are relevant to this lab.

Your files and directories exist in a *filesystem*. Linux has no concept of 'drive letters' as in Windows; instead all files, folders and other objects such as symbolic links (similar to a Windows shortcut) are part of a single *filesystem tree*. The *root* of this tree is represented with a single forward slash (/) character. This contains many directories (folders), such as `/bin` (where many basic Linux programs are stored), `/home` (where your *home directory* is located), `/tmp` (which is a folder you can use for temporary storage) and `/lib` (used to store important shared library files - equivalent to Windows DLLs). The most important of these (for you) is your home directory. If your login is `123456`, then your home directory will (normally) be `/home/123456`. When you open a shell (terminal window) on Linux, this is normally where you will 'be' by default, and it is where all your personal files and directories are stored. You can list these by typing `ls` (list) at the command prompt. As shorthand for your home directory, you can use the ~ character; for example, `cd ~/comm-protocols/labs` takes you to the folder `comm-protocols-labs` inside your home directory from anywhere in the system.

Some other useful commands:

- `cd` *`directory`*: change to specified directory. To go up one level you can type `cd ..` (two dots).
- `mkdir [-p]` *`directory`*: create an empty folder. -p allows you to create a cascade of folders (e.g. `mkdir -p folder/with/other/folders/in/it`
- `rm [-r]` *`files/dirs`*: delete a file. `-r` is recursive delete - you can use it to delete a directory and all of its contents
- `ln [-s] targetpath alias`: create a symbolic link called `alias` which points to `targetpath`. `-s` makes a symbolic link (default is hard links which are not very useful).
- `rmdir` *`directory`*: deletes an empty directory
- `ps [aux]`: process status - see what is running. Note: you don't need a - for the options in this case. a = all, u = user-oriented format, x = processes not started from command line
- `gedit`: graphical text editor

A really helpful tip: the Linux shell supports **tab completion**. If you type the first few characters of a command or a filename, followed by the `Tab` key, the shell will complete the rest for you. This saves

a lot of typing. For example:

```
cd ~/com[Tab]
```

will expand to `cd ~/comm-protocols-labs`.

3.1 Job and Process Control

In Linux, a program that is either running or paused is called a *process*. This includes applications started by users, and system processes such as the web server or DNS server. Each process has a process ID number (PID), with PID 1 being the very first process started when the operating system starts (known as the *init* process), and all other processes being sequentially numbered in order of creation

A process that is started interactively from a command interpreter shell (such as the one running in your terminal, into which you type commands) is known as a *job*. Each shell maintains its own list of jobs, each of which is given a number starting at 1.

When you start an application from the command line, such as a mininet script or a graphcial text editor, it runs in the 'foreground'. This means that it blocks further input from the shell where you started the command. If you try to type a shell command in the terminal, it won't work - the characters can only be read by the running foreground process. Often, you would like to get your shell back, but allow the process to continue running in the background. Suppose we want to start the gedit graphical editor. We can start it like this:

```
gedit &
```

(note the & character at the end) to start it in the background. The shell will now display something like this:

```
[1] 143616
```

and you may continue typing Linux shell commands as before, while the editor will run normally.

The number in the square brackets is the *job number* - the number of the background job started from this particular shell - while the other number is a global process ID number whcih can be used to refer to this process (the graphcial editor) from anywhere. If you want to bring this job back to the foreground, you may type `fg %1` (or actually even just `fg` since there's only one background job). You may also start a process without the & character and put it into the background. If you had just typed `gedit` and then switched back to the terminal, you can press `ctrl-z` to *suspend* the editor (which will essentially freeze it - the editor's GUI will become unresponsive) and then type `bg %1` or `bg` to put the suspended job in the background (if the shell says the job number is something other than 1, use that number. If you aren't sure about the job numbers, the `jobs` command will tell you.

The second number (143616 in the example above) can be used to control processes from anywhere (e.g. another terminal). You do this by sending signals with the `kill` command. For example, you can suspend (pause) a process by typing

```
kill -STOP 143616
```

If you want to kill a process that's gotten into a weird state, you may use `kill -9 processID` for an unconditional process termination. If you aren't sure what the ID of a particular process is, you can use the `ps aux` command (process status, with the options a = all, u = user-oriented output format, x = all processes including those not started interactively) to see the process ID numbers associated with particular process names. `kill` also works with job numbers, e.g. `kill %3` to kill background job 3.

You can also use the convenient `killall` command, for example:

```
killall gedit
```

will send a shutdown signal to all gedit processes (use `killall -9 gedit` if that doesn't work).

The `top` command is also useful to see what processes are using lots of CPU or memory. You can type `m` while top is running to sort processes in order of memory usage.

Most foreground jobs under Linux can be (gracefully) killed by pressing `ctrl-c`. If that doesn't work, you can try `ctrl-z` to suspend (pause) it and then `kill -9 %1` to kill it (using whatever job number is appropriate instead of %1).

References

[1] James F. Kurose and Keith W. Ross. *Computer Networking A Top-Down Approach*. 8th edition, 2020.

[2] Behrouz A. Forouzan. *TCP/IP Protocol Suite*. 4th edition, 2017.