Understanding Linux Networking Lab Workbook

P2000

Lab Environment Review

Host Laptop Review

Host OS: Ubuntu 10.04 LTS
Host Linux Kernel: Version 3.1.6

Host Laptop Login information

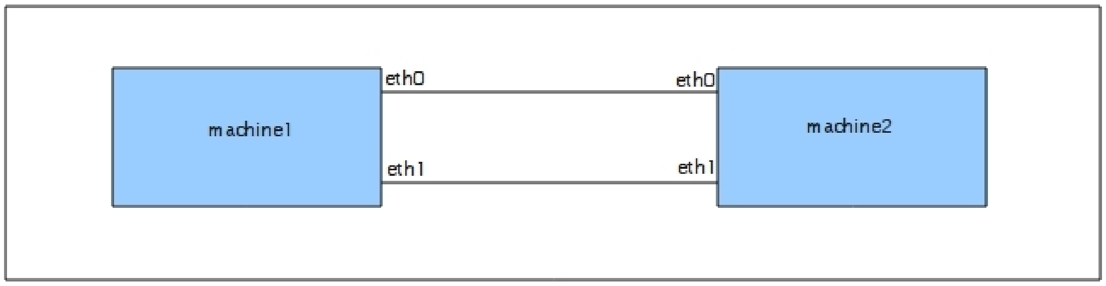
Login Name: ***student***	(Instructor assigned for on-line classes)
Password: ***student***	(Instructor assigned for on-line classes)
Home Directory: ***/home/student***	

Target LAN Review

The target LAN that we will use for our networking labs consists of two PCs that run as virtual machines on your host laptop. These PCs are networked together and behave as physical PCs on a LAN.

Target PC OS: Xubuntu 12.04 LTS
Target PC Linux Kernel: Version 3.2.0

The two PC virtual machines run as “machine1” and “machine2”.
Interface eth0 on machine1 is connected to eth0 on machine2.
Interface eth1 on machine1 is connected to eth1 on machine2.
These are two independent Ethernet connections.



Target PC LAN

We will see how to start up the target LAN in the first networking lab.

This completes the lab environment review. If you have any additional questions about the setup, feel free to ask the instructor. Enjoy your lab work!

Lab 1. Networking and Linux Boot

Objective

Explore the networking aspects of the Linux boot sequence.

Procedure

The first step we will perform is to start the target PC LAN. Open a terminal window on your training laptop by clicking the Terminal icon in the top border of the desktop.

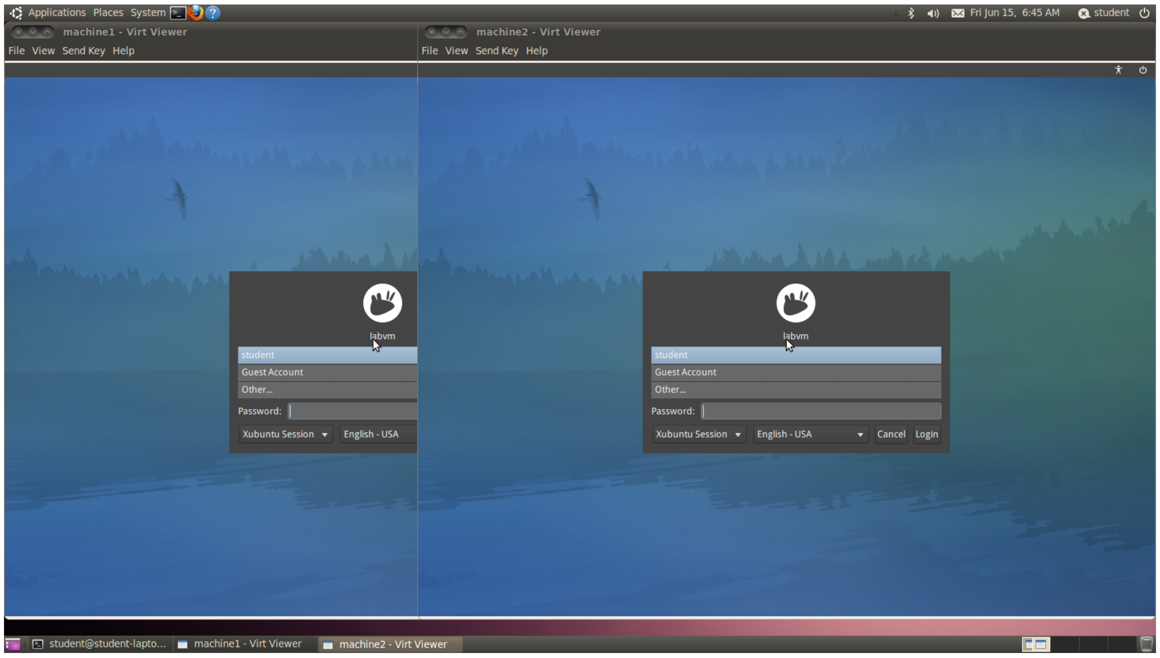
In the terminal window, start the target PC LAN by issuing the **runlan** command.

```
$ runlan
```

Bringing up the target PC LAN requires sudo privileges, so you will be prompted for your host laptop login password. Enter the password that you entered when logging into the laptop.

Note that each time you enter the password and are authenticated for sudo privileges, you may then do sudo operations for a period of time without entering the password again. After this period of time, the next sudo operation you perform will prompt you for the password again.

The **runlan** command starts up the two target PCs as machine1 and machine2. Click the top border of machine2 and drag it to the right part of the laptop display, and drag machine1 to the left part of the display.

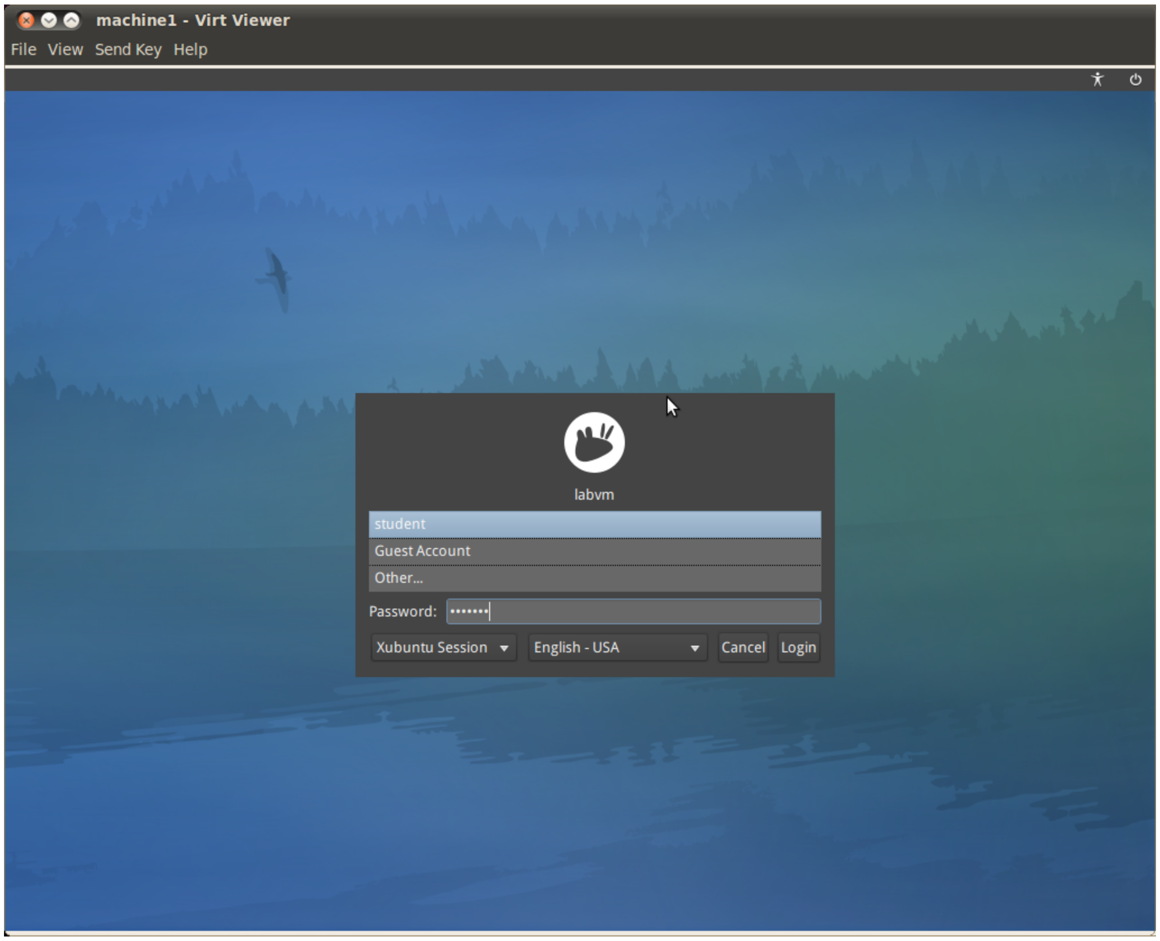


Give the target PCs some time to initialize before attempting to login. As the target PCs are loading into memory from the hard drive, they will be slow to respond. But after a few moments the target PCs have booted and display their login screens. The target PCs use username “**student**” and password “**student**”. Remember that these two machines are two complete PCs on a LAN with one another.

It is very important to understand that these target PCs receive fresh, new disk images each time we use the **runlan** command. This means that any changes we make on the target PCs are gone the next time we use **runlan**. This is a good thing, as it allows us to set up very interesting networking configurations and investigate different settings, while not risking the functionality of the later labs. However, if there is anything you want to save, you need to copy it off of the target VMs before exiting. We’ll show you how to do that in a later lab.

So, at the end of every lab, we will shut down both target PCs. And, at the beginning of the following lab, we will start fresh target PCs by using the **runlan** command.

Bring machine1 to the foreground by clicking on its top window border. Login to machine1 by making sure that username “student” is highlighted, then clicking on the “Password:” entry box and entering the password “**student**”.



Start a terminal window on machine1 by right-clicking on its desktop background, and selecting "Open Terminal Here". Click on the bottom border of the new terminal and drag it down to make it larger.

Now we will look at the output of the "dmesg" command, which displays the kernel log buffer. This is a circular buffer that displays the latest kernel log output.

You can use the mouse to drag the scroll bar on the right edge of the window to scroll up and down through the text, but the resulting output is several hundred lines of text and may exceed the default scrolling capability of the terminal. One way to work around this is to pipe the output of "dmesg" into a program like "less" as follows:

```
    $ dmesg | less
```

Enter the above command in your terminal window on machine1.

Note that the following output may look different on your machine, and may appear differently on the virtual training laptop versus the physical training laptop.

The important thing is to look carefully at your dmesg output to see what machine1 is telling you about its boot-up sequence.

We will focus on the log output related to networking. Scroll down past the initial CPU startup messages. Early in the boot sequence we see the Netlink protocol being registered.

```
...
[    0.032001] NET: Registered protocol family 16
```

Below that the Internet IP protocol is registered, followed by the setup of hash tables for IP, TCP, and UDP. The TCP "reno" congestion control algorithm is registered with the network stack. Then, the Unix-domain socket protocol is registered.

```
...
[    0.112231] NET: Registered protocol family 2
[    0.112298] IP route cache hash table entries: 32768 (order: 5, 131072 bytes)
[    0.112564] TCP established hash table entries: 131072 (order: 8, 1048576 bytes)
[    0.113364] TCP bind hash table entries: 65536 (order: 7, 524288 bytes)
[    0.113746] TCP: Hash tables configured (established 131072 bind 65536)
[    0.113750] TCP reno registered
[    0.113752] UDP hash table entries: 512 (order: 2, 16384 bytes)
[    0.113765] UDP-Lite hash table entries: 512 (order: 2, 16384 bytes)
[    0.113846] NET: Registered protocol family 1
```

The tunneling device driver is initialized.

```
...
[    0.404445] tun: Universal TUN/TAP device driver, 1.6
[    0.404446] tun: (C) 1999-2004 Max Krasnyansky <maxk@qualcomm.com>
```

The PPP driver is initialized.

```
...
[    0.580551] PPP generic driver version 2.4.2
```

The TCP cubic congestion control algorithm is registered with the network stack. Then the IPv6 protocol is registered, followed by the raw packet protocol.

```
...
[    0.585424] TCP cubic registered
[    0.585544] NET: Registered protocol family 10
[    0.586149] NET: Registered protocol family 17
```

The kernel registers the dns_resolver key type, which is a method the kernel can use to cause user space helper tools to perform DNS queries.

```
...
[    0.586169] Registering the dns_resolver key type
```

The next messages come from an early call to IPv6 addressing code for the eth0 and eth1 interfaces.

```
...
[    1.591292] ADDRCONF(NETDEV_UP): eth0: link is not ready
[    1.591298] ADDRCONF(NETDEV_UP): eth1: link is not ready
```

The last network-related bootup message shows the Bluetooth protocol being registered.

```
...
[    3.817961] NET: Registered protocol family 31
```

Our Linux system contains file systems /proc and /sys that are windows into the dynamic state of the system. This includes kernel and network stack information. For example, we can look at the file entries in /sys/class/net/eth0/ to determine the state of our eth0 interface:

```
student@labvm:~$ ls /sys/class/net/eth0
addr_assign_type  device   ifalias    netdev_group  statistics
address           dev_id   ifindex    operstate     subsystem
addr_len          dormant  iflink     power         tx_queue_len
broadcast         duplex   link_mode  queues        type
carrier           flags    mtu        speed         uevent
```

The /proc/net directory contains current statistics and state information:

```
student@labvm:~$ ls /proc/net
anycast6      icmp          ip_mr_vif    protocols  rt_cache      tcp6
arp           if_inet6      ipv6_route   psched     snmp          tr_rif
connector     igmp          mcfilter     ptype      snmp6         udp
dev           igmp6         mcfilter6    raw        sockstat      udp6
dev_mcast     ip6_flowlabel netfilter    raw6       sockstat6     udplite
dev_snmp6     ip6_mr_cache  netlink      route      softnet_stat  udplite6
fib_trie      ip6_mr_vif    netstat      rt6_stats  stat          unix
fib_triestat  ip_mr_cache   packet       rt_acct    tcp           wireless
```

The /proc/sys/ directory has entries that actually let us view and control many features of the running system. The /proc/sys/net/ directory contains the trees of control for the network stack.

```
student@labvm:~$ ls /proc/sys/net
core  ipv4  ipv6  netfilter  token-ring  unix
```

For example, let's look at the items available in /proc/sys/net/ipv4:

```
student@labvm:~$ ls /proc/sys/net/ipv4
cipso_cache_bucket_size            ping_group_range                  tcp_mtu_probing
cipso_cache_enable                 route                             tcp_no_metrics_save
cipso_rbm_optfmt                   rt_cache_rebuild_count            tcp_orphan_retries
cipso_rbm_strictvalid              tcp_abc                           tcp_reordering
conf                               tcp_abort_on_overflow             tcp_retrans_collapse
icmp_echo_ignore_all               tcp_adv_win_scale                 tcp_retries1
icmp_echo_ignore_broadcasts        tcp_allowed_congestion_control    tcp_retries2
icmp_errors_use_inbound_ifaddr     tcp_app_win                       tcp_rfc1337
icmp_ignore_bogus_error_responses  tcp_available_congestion_control  tcp_rmem
icmp_ratelimit                     tcp_base_mss                      tcp_sack
icmp_ratemask                      tcp_congestion_control            tcp_slow_start_after_idle
```

```
igmp_max_memberships              tcp_cookie_size                 tcp_stdurg
igmp_max_msf                      tcp_dma_copybreak               tcp_synack_retries
inet_peer_maxttl                  tcp_dsack                       tcp_syncookies
inet_peer_minttl                  tcp_ecn                         tcp_syn_retries
inet_peer_threshold               tcp_fack                        tcp_thin_dupack
ip_default_ttl                    tcp_fin_timeout                 tcp_thin_linear_timeouts
ip_dynaddr                        tcp_frto                        tcp_timestamps
ip_forward                        tcp_frto_response               tcp_tso_win_divisor
ipfrag_high_thresh                tcp_keepalive_intvl             tcp_tw_recycle
ipfrag_low_thresh                 tcp_keepalive_probes            tcp_tw_reuse
ipfrag_max_dist                   tcp_keepalive_time              tcp_window_scaling
ipfrag_secret_interval            tcp_low_latency                 tcp_wmem
ipfrag_time                       tcp_max_orphans                 tcp_workaround_signed_windows
ip_local_port_range               tcp_max_ssthresh                udp_mem
ip_local_reserved_ports           tcp_max_syn_backlog             udp_rmem_min
ip_nonlocal_bind                  tcp_max_tw_buckets              udp_wmem_min
ip_no_pmtu_disc                   tcp_mem                         xfrm4_gc_thresh
neigh                             tcp_moderate_rcvbuf
```

There are many entries there that provide visibility and control of numerous networking features. One key feature that you will encounter from time to time is "**ip_forward**". That is a switch that allows us to control whether or not our Linux system is capable of forwarding packets between interfaces. Enabling this allows our system to operate as a gateway or router.

Shut down both of your target PCs. Do this by typing "exit" in any open terminal windows in the target PCs until there are no remaining terminal windows open. Then in the upper right of the target PC window (**not** the overall host laptop screen) click on the word "student", pull that menu down, and choose "Shut down". Click "Shut down" in the popup that appears. Alternatively if the target PC is still at its initial login screen, you can go to the top right of its window, click on the icon that looks like a power button, and choose "Shutdown…" and then confirm by clicking "Shutdown" in the pop-up that appears.

Lab Completed.

Lab 2. Digging into a Driver

Objective

Examine a working driver for key features.

Procedure

In this lab, we will be working on the host computer to examine several features of network drivers that were discussed in class. First, open a terminal on the host system by clicking the Terminal icon in the top border of the desktop. Then change directory to ~student/network-labs/lab02.

```
$ cd ~student/network-labs/lab02
```

Next, we will extract a version of the Linux kernel from the .tar.xz file located in the directory. .xz files are compressed with a newer compression algorithm that provides somewhat better performance than either .gz or .bz2 style compression. Once the kernel is extracted, cd into the network driver's directory for the Intel e1000e driver.

```
$ tar Jxf linux-3.2.72.tar.xz
$ cd ./linux-3.2.72/drivers/net/ethernet/intel/e1000e
```

Load the **`netdev.c`** file into the editor of your choice. Once the file is loaded into the editor, find the **`e1000_probe`** function. This is where the network driver is registered with the kernel. Find the allocation function for the netdev structure. As discussed in class, this would be either an **`alloc_etherdev`** or an **`alloc_netdev`** call.

Which is it in this driver? alloc_etherdev

Since this is a real driver, we see a lot of the specifics for the hardware in this code that we did not discuss in class. For example, the **`SET_NETDEV_DEV(netdev, &pdev->dev)`** call is used to associate the newly allocated netdev structure with the internal PCI bus device structure used by the Linux driver model (LDM) to track physical devices.

Next, find the line of code where we are storing the callback address of the netdev_ops callbacks and enter that text here:

netdev->netdev_ops = &e1000e_netdev_ops;

The next line, **`e1000e_set_ethtool_ops(netdev)`** is where we work with the PHY and make the connection with the layer 2 **`ethtool`** command although the actual PHY driver code is located in the file **`phy.c`**.

Next, locate the name of the callback associated with the **`ndo_start_xmit`** function and enter it here:

e1000e_netdev_ops

In the **`ndo_start_xmit`** callback code, the driver uses the hardware to calculate the packet checksum. What is the name of the function that is used to set the hardware up to do the actual checksum calculation for a transmit packet? Enter its name here:

e1000_tx_csum()

Does this driver support changing the size of the MTU for the device (Y/N)?
If yes, what is the name of the function that if responsible for changing the MTU?

e1000_change_mtu()

Extra Credit:

This driver also uses a DMA ring buffer to transmit potentially several packets at once to improve performance. What is the name of the function that handles the queuing of the packets into the transmit ring? Enter it here:

e1000_tx_queue()

Lab Completed.

Lab 3. IPv6 Networking

Objective

Configure IPv6 address assignments and establish IPv6 communications.

Procedure

Part A.

Be sure to start with the target PC LAN VMs completely shut down. We want to have a known configuration at the start of each lab.

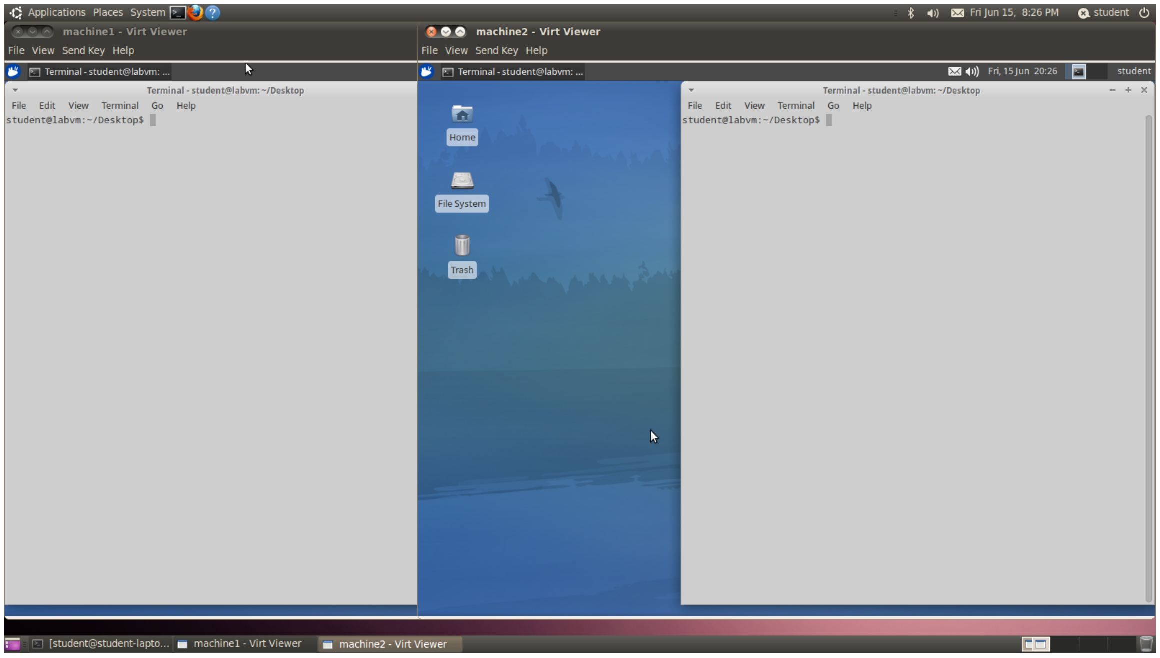
In the terminal window on your host laptop, start a new target PC LAN by issuing the **runlan** command.

```
$ runlan
```

Remember to click the top border of machine2 and drag it to the right part of the laptop display, and drag machine1 to the left part of the display.

Login as the "student" user on both the machine1 and machine2 VMs. Start terminal windows on both machine1 and machine2 by right-clicking on each of their desktop's background, and selecting "Open Terminal Here". Click on the bottom border of the new terminals and drag down to make them larger.

You can show both the machine1 terminal and the machine2 terminal by moving and sizing the terminals as shown here.



```
student@labvm:~/Desktop$
```

```
student@labvm:~/Desktop$
```

Many of the steps that we will perform in these labs are system configuration procedures that require us to have root privileges on the target PC. The target PC is set up for us to be able to do this by using the command “sudo su –“. Be sure to type the hyphen at the end. You will be prompted for the student password, which on the target PC machines is “student”.

```
$ sudo su –
[sudo] password for student:
root@labvm:~#
```

Do this in the terminals on machine1 and machine2.

Now that you are running as root, on each target PC, do “iip link set dev eth0 up”. Then type “ip addr show dev eth0”. You will see that the eth0 interfaces on the two machines have different MAC addresses (shown as HWaddr). Here is the output for machine1.

```
# ip link set dev eth0 up
# ip addr show dev eth0
2: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc
pfifo_fast state UP qlen 1000
    link/ether 52:54:00:45:2d:f0 brd ff:ff:ff:ff:ff:ff
    inet6 fe80::5054:ff:fe45:2df0/64 scope link

       valid_lft forever preferred_lft forever
```

Be sure to do the same on both target PCs. You will also see that each eth0 interface has a unique default link local IPv6 address starting with “fe80”. If you look closely at the IPv6 link local address on each interface, you will see the similarities with the interface's MAC address.

On machine2, let's ping machine1 using its IPv6 address. Perform the following on machine2.

```
# ping6 –I eth0 fe80::5054:ff:fe45:2df0
```

Notice that we used the "ping6" command to use IPv6. We also have to specify the interface "eth0". There are two ways to specify the interface. The first way is to use the "`-I`" option as shown above. The alternate way is to specify "%eth0" at the end of the IPv6 address as shown here:

```
# ping6 fe80::5054:ff:fe45:2df0%eth0
```

Now we will use ssh to connect from machine1 to machine2 using IPv6 networking. With ssh, we need to specify the interface using the "%eth0" suffix on the IPv6 address.

Go to machine1 and do the following.

```
# ssh student@fe80::5054:ff:fe45:2df2%eth0
The authenticity of host 'fe80::5054:ff:fe45:2df2%eth0
(fe80::5054:ff:fe45:2df2%eth0)' can't be established.
ECDSA key fingerprint is ce:e8:a8:64:56:92:0f:cb:7e:f5:1f:24:53:0a:ee:8b.
Are you sure you want to continue connecting (yes/no)? yes
Warning: Permanently added 'fe80::5054:ff:fe45:2df2%eth0' (ECDSA) to the
list of known hosts.
student@fe80::5054:ff:fe45:2df2%eth0's password:
Welcome to Ubuntu 12.04 LTS (GNU/Linux 3.2.0-25-generic i686)

 * Documentation:  https://help.ubuntu.com/


The programs included with the Ubuntu system are free software;
the exact distribution terms for each program are described in the
individual files in /usr/share/doc/*/copyright.

Ubuntu comes with ABSOLUTELY NO WARRANTY, to the extent permitted by
applicable law.

student@labvm:~$
```

Since this is the first ssh from machine1 to machine2 in this context, you will be prompted to accept the digital fingerprint before you can continue. Type "yes". Then at the password prompt enter the student password. You are now logged into machine2 via secure shell. Note that both machine1 and machine2 use the hostname "labvm". But, you can tell that you are on machine2 by doing "ip link show eth0" and looking at the MAC address. The eth0 MAC address ending with byte "f2" belongs to machine2.

Type "exit" to leave the ssh session and return to machine1. Exit all target PC terminals, and completely shut down both target PCs (machine1 and machine2).

Part B

In part B of this lab, we will be completing a client/server application pair that uses IPv6. Before we do that, let's make file transfer a bit easier between the host and the VMs. Switch back to machine1 and enter:

```
# ip addr add 10.1.1.1/24 dev eth0
# ip link set dev eth0 up
```

While on machine1, open another terminal window and as the student user create a lab03 directory for the destination of the lab code:

```
$ mkdir ~/lab03
```

And, open a terminal window on the development host (where you ran the "runlan" command) and enter:

```
$ sudo ip addr add 10.1.1.2/24 dev br0
# ip link set dev br0 up
```

This will add an IPv4 address to the LAN bridge between the two environments to make it easier to type in the copy commands. Next, on the host system, cd to the lab03 directory:

```
$ cd ~student/network-labs/lab03
```

Edit the **`server.c`** code to fill in the missing pieces based on the material found in the sockets chapter. You can run the compiler to verify the code is syntactically correct using the "make" command in that directory. Iterate with the code to make sure that you get it to compile. Once the code is compiling, we can test the code on the development host. This is best viewed with two terminal windows.

On window one:

```
$ ./server <unused port number>
```

You can look at the **`/etc/services`** file for a quick listing of allocated ports. But, a port like 1929 should work in most cases.

On window two:

```
$ ./client ::1 <server port number>
```

If your code is working, you should see something like this on window 1:

```
$ ./server 1929
```

```
IPv6 TCP Server Started...
Ready for client connect().
Client address is ::1
Client port is 51841
Message from client: This is a message from the client!
```

On the window 2, you should see something like this:

```
./client ::1 1929

IPv6 TCP Client Started...
Message from server: Server got your message
```

Next, we move the code to machine1. Since machine1 is running an **ssh** server, the easy way to move the code is to use the **scp** program. From window one where you ran the server, enter:

```
$ scp * student@10.1.1.1:/home/student/lab03
```

You may be asked for the student ID password. This should copy all of the code to the lab03 directory that you created earlier. Next, cd into that directory on the machine1 and enter:

```
$ make clean; make
```

This will rebuild the code for the machine1.

Back on the development host, enter:

```
$ ip -6 addr show dev br0

7: br0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500
    inet6 fe80::fc54:ff:fe45:2fd0/64 scope link
       valid_lft forever preferred_lft forever
```

Your IPv6 address may be different. Next, we run the server on the development host and start the client on machine1.

On the development host window 1:

```
$ ./server <unused port number>
```

On machine1 enter:

```
$ ./client <IPv6 address of br0>%eth0 <server port number>
```

Remember to use the special <IPv6 addr>%eth0 syntax on the client or the client will fail to connect. If this fails, make sure that the server is running on the development host, that you have the correct IPv6 address from the bridge and that you have the same port number on both ends.

If everything is successful, you should see output like that from the test on the development host except that the IPv6 address is that of machine1's eth0 (fe80::5054:ff:fe45:2df0). If not, keep trying and make sure that you have all of the addresses (especially the IPv6 addresses) typed exactly as they appear in the **`ip addr show`** command output.

Make sure that you shut down all of the VMs and close any extraneous windows on the development host to clean up before moving to the next lab.

Lab Completed.

Lab 4. Quagga

Objective

Configure and run Quagga. Observe the routing daemons' operational behavior.

Procedure

Start with the target PC LAN completely shut down.

In the terminal window on your host laptop, start a new target PC LAN by issuing the **runlan** command. Once the target PCs are up and running, remember to click the top border of machine2 and drag it to the right part of the laptop display, and drag machine1 to the left part of the display.

Start terminal windows on both machine1 and machine2, and become root on both target PCs by using the "sudo su –" command.

In this lab, the target PCs will communicate with each other over their eth0 interfaces.

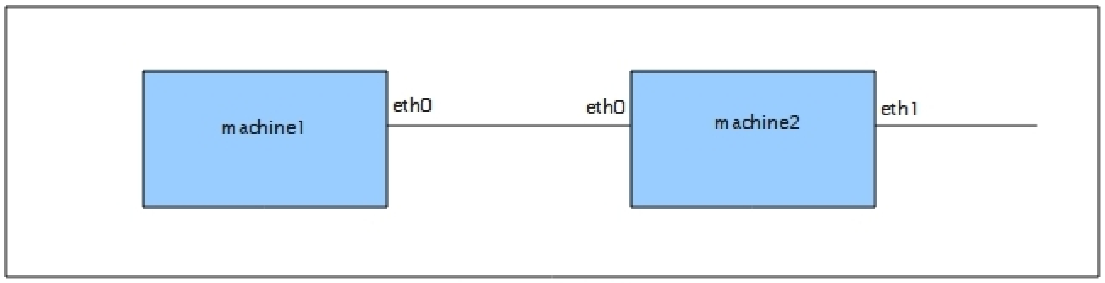
On machine1 bring up eth0 with IP address 99.99.99.10.

```
# ip addr add 99.99.99.10/8 dev eth0
# ip link set dev eth0 up
```

On machine2 bring up eth0 with IP address 99.99.99.11.

```
# ip addr add 99.99.99.11/8 dev eth0
# ip link set dev eth0 up
```

In addition, we will use the eth1 interface on machine2. We will not bring up eth1 on machine1. The network layout used in this lab is as follows.



Target PC LAN (Quagga Lab)

On machine2 bring up eth1 with IP address 100.100.100.11.

```
# ip addr add 100.100.100.11/8 dev eth1
# ip link set dev eth1 up
```

Make sure that you do not bring up eth1 on machine1.

Next, we will configure quagga by setting up the zebra and ospfd daemon processes that run as part of quagga.

The following steps need to be completed identically on **both** machine1 and machine2:

1. Copy sample config files into place for zebra and ospfd. Notice the filenames change.

   ```
   # cp /usr/share/doc/quagga/examples/zebra.conf.sample /etc/quagga/zebra.conf
   # cp /usr/share/doc/quagga/examples/ospfd.conf.sample /etc/quagga/ospfd.conf
   ```

2. Edit file **`/etc/quagga/daemons`**. Change the lines for zebra and ospfd from no to **yes**. The uncommented lines in that file should now look like this on both machine1 and machine2.

   ```
   zebra=yes
   bgpd=no
   ospfd=yes
   ospf6d=no
   ripd=no
   ripngd=no
   isisd=no
   ```

3. Edit file **`/etc/quagga/ospfd.conf`**. Uncomment the "router ospf" section, and fill it in as shown below. Be sure that you set up **`ospfd.conf`** on both machines identically. This will set up the ospfd daemons to communicate over the 99.0.0.0/8 eth0 network, and to redistribute to each other all route information about connected interfaces, and routes learned by the kernel. The file should appear as follows.

```
! -*- ospf -*-
!
! OSPFd sample configuration file
!
!
hostname ospfd
password zebra
!enable password please-set-at-here
!
router ospf
  network 99.0.0.0/8 area 0
  redistribute connected
  redistribute kernel
!
log stdout
```

4. Run **/etc/init.d/quagga restart**.

```
# /etc/init.d/quagga restart
```

Verify that you have performed the numbered list of instructions above on both machine1 and machine2.

Next, go to machine1. We will watch as machine1 learns routes from machine2. On machine1 enter the following command.

```
# watch route -n
```

The watch command will execute "route –n" every 2 seconds, and keep refreshing the display of its output. Arrange your display such that you can observe the output of the "watch" command on machine1, while entering commands on machine2.

When we first enter the "watch" command above on machine1, the output will appear as follows. Note that this is temporary and will only appear this way until quagga's daemons have communicated.

```
Every 2.0s: route -n                                  Sat Jun 16 09:26:46 2015

Kernel IP routing table
Destination     Gateway         Genmask         Flags Metric Ref    Use Iface
99.0.0.0        0.0.0.0         255.0.0.0       U     0      0        0 eth0
```

This is the routing table on machine1. At this point it only knows about its own route on its eth0 interface.

However, once the quagga daemons communicate, you will see machine1 start to learn routes from machine2 automatically, as shown here.

```
Every 2.0s: route -n                                  Sat Jun 16 09:27:04 2015
```

```
Kernel IP routing table
Destination     Gateway         Genmask         Flags Metric Ref    Use Iface
99.0.0.0        0.0.0.0         255.0.0.0       U     0      0        0 eth0
100.0.0.0       99.99.99.11     255.0.0.0       UG    20     0        0 eth0
```

Notice that this happens automatically. Look at the new last line in the output above. The daemons have communicated, and machine1 now knows how to reach network 100.0.0.0/8 (the "/8" is another way of writing the information in the mask field above). This network is actually on machine2's eth1 interface, but machine1 has learned that it can **reach** this network by sending to 99.99.99.11 (machine2's IP address on eth0) as a gateway (note the 'G' flag). And machine1 knows that it will need to use its own eth0 interface to reach this gateway (as seen in the new route's "Iface" field).

Now, we will add more routes on machine2, and watch as machine1 learns them. On machine2 issue the following command.

```
# ip route add 123.0.0.0/8 dev eth1
```

Observe the output on machine1. The new route is learned within a few seconds.

```
Every 2.0s: route -n                                   Sat Jun 16 09:27:20 2015

Kernel IP routing table
Destination     Gateway         Genmask         Flags Metric Ref    Use Iface
99.0.0.0        0.0.0.0         255.0.0.0       U     0      0        0 eth0
100.0.0.0       99.99.99.11     255.0.0.0       UG    20     0        0 eth0
123.0.0.0       99.99.99.11     255.0.0.0       UG    20     0        0 eth0
```

On machine2 issue the following command (we're using the older syntax to show you how that works as well):

```
# route add -net 145.12.0.0/16 dev eth1
```

Notice the difference in the netmask size.

Observe the output on machine1. The netmask size is reflected correctly.

```
Every 2.0s: route -n                                   Sat Jun 16 09:28:12 2015

Kernel IP routing table
Destination     Gateway         Genmask         Flags Metric Ref    Use Iface
99.0.0.0        0.0.0.0         255.0.0.0       U     0      0        0 eth0
100.0.0.0       99.99.99.11     255.0.0.0       UG    20     0        0 eth0
123.0.0.0       99.99.99.11     255.0.0.0       UG    20     0        0 eth0
145.12.0.0      99.99.99.11     255.255.0.0     UG    20     0        0 eth0
```

And finally we will add a new host route on machine2.

```
# ip route add 156.98.17.44/32 dev eth0
```

Observe the output on machine1. Notice the 'H' flag on the new host route.

```
Every 2.0s: route -n                                       Sat Jun 16 09:29:02 2015

Kernel IP routing table
Destination     Gateway         Genmask         Flags Metric Ref    Use Iface
99.0.0.0        0.0.0.0         255.0.0.0       U     0      0        0 eth0
100.0.0.0       99.99.99.11     255.0.0.0       UG    20     0        0 eth0
123.0.0.0       99.99.99.11     255.0.0.0       UG    20     0        0 eth0
145.12.0.0      99.99.99.11     255.255.0.0     UG    20     0        0 eth0
156.98.17.44    99.99.99.11     255.255.255.255 UGH   20     0        0 eth0
```

Exit all terminals on the target PCs. Shut down the target PCs completely.

Lab Completed.

Lab 5. Ethernet Bonding

Objective

Configure and run multiple Ethernet interfaces in a bonded interface.

Procedure

We have learned that an Ethernet bond can run in different modes to serve different purposes. In this lab, we will run our bonded interface in “balance-rr” mode, which is a performance enhancement mode.

Start with the target PC LAN completely shut down.

In the terminal window on your host laptop, start a new target PC LAN by issuing the **runlan** command. Arrange the target PC windows as you have done in the previous labs so you can see both outputs.

Start terminal windows on both machine1 and machine2, and become root on both target PCs by using the “sudo su –“ command. Also, we will use the older ifconfig syntax to give you a taste of the original API. For an extra challenge, using the bonding chapter as a guide, substitute the newer API using the “ip” command.

Perform the following commands on **machine1** only.

```
# modprobe bonding
# ifconfig bond0 up
# ifenslave bond0 eth0 eth1
# ifconfig bond0 192.168.10.10
# cat /sys/class/net/bond0/bonding/mode
```

You have loaded the bonding kernel module on machine1, and enslaved the eth0 and eth1 interfaces to the new bond0 interface. You gave bond0 an IP address. Finally, you checked the bonding mode of bond0, which will be displayed as “`balance-rr 0`”.

Next, on machine1 we will run an iperf server. The iperf tool is used to measure network performance. Do the following on **machine1**:

```
# iperf -s
```

Now go to **machine2** and perform the following commands.

```
# modprobe bonding
# ifconfig bond0 up
# ifenslave bond0 eth0 eth1
# ifconfig bond0 192.168.10.11
```

Notice the different IP address for the bond on machine2.

Our target PCs run in virtual machines, which means that the Ethernet interfaces between them run at extremely high speed, since they are virtual, and not attached to physical Ethernet cables. In order for us to show the bonding behavior on real/physical Ethernet interconnects, we need to run a special command script that will perform rate limiting on eth0 and eth1 on machine2. This script will cause eth0 and eth1 on machine2 to run at around 5 Mbit/sec each. This is meant to reduce the speed numbers down to a rate where we are not bounded by CPU speed, but rather by a virtual bandwidth limitation. The script uses a traffic control tool "tc" that you will learn about in class.

Do this command only on **machine2**.

```
# /home/student/scripts/bonding-lab/bond-traffic-setup.sh
```

Now on machine2, run the iperf client, which will communicate with the iperf server that we started on machine1.

Do this command on **machine2**.

```
# iperf –c 192.168.10.10 –i 1 –t 10000
```

This will test the throughput of bond0, outputting traffic speed every second. It will run for 10000 seconds unless stopped via <ctrl>-c.

Look at the traffic speed through bond0. It should look similar to this.

```
[  3] local 192.168.10.11 port 45392 connected with 192.168.10.10 port 5001
[ ID] Interval       Transfer     Bandwidth
[  3]  0.0- 1.0 sec  1.25 MBytes  10.5 Mbits/sec
[  3]  1.0- 2.0 sec  1.12 MBytes  9.44 Mbits/sec
[  3]  2.0- 3.0 sec  1.12 MBytes  9.44 Mbits/sec
[  3]  3.0- 4.0 sec  1.12 MBytes  9.44 Mbits/sec
[  3]  4.0- 5.0 sec  1.25 MBytes  10.5 Mbits/sec
[  3]  5.0- 6.0 sec  1.12 MBytes  9.44 Mbits/sec
```

Next on machine2 we will remove eth0 from the bond, and watch the performance change.

Start a new terminal on machine2 and become root in it using "sudo su –". In this new terminal on machine2, do the following command.

```
      # ifenslave –d bond0 eth0
```

Now that eth0 is deleted from the bond, notice the iperf performance drop. This is because the bond has lost one of its two interfaces, and is running in balance-rr mode.

```
    [  3] 63.0-64.0 sec   640 KBytes  5.24 Mbits/sec
    [  3] 64.0-65.0 sec   640 KBytes  5.24 Mbits/sec
    [  3] 65.0-66.0 sec   512 KBytes  4.19 Mbits/sec
    [  3] 66.0-67.0 sec   640 KBytes  5.24 Mbits/sec
    [  3] 67.0-68.0 sec   512 KBytes  4.19 Mbits/sec
    [  3] 68.0-69.0 sec   640 KBytes  5.24 Mbits/sec
    [  3] 69.0-70.0 sec   512 KBytes  4.19 Mbits/sec
```

Next, we will add eth0 back into the bond.

Perform the following command on machine2.

```
      # ifenslave bond0 eth0
```

Observe the performance change, now that the bond is running with two interfaces again.

```
      [  3] 125.0-126.0 sec  1.12 MBytes  9.44 Mbits/sec
      [  3] 126.0-127.0 sec  1.12 MBytes  9.44 Mbits/sec
      [  3] 127.0-128.0 sec  1.12 MBytes  9.44 Mbits/sec
      [  3] 128.0-129.0 sec  1.12 MBytes  9.44 Mbits/sec
      [  3] 129.0-130.0 sec  1.25 MBytes  10.5 Mbits/sec
```

This shows the behavior of balance-rr mode. The bond uses its interfaces in a balanced round robin fashion to get maximum throughput.

Exit all terminals on the target PCs. Shut down the target PCs completely.

Lab Completed.

Lab 6. Quality of Service (QoS)

Objective

Configure QoS to control network traffic.

Procedure

Start with the target PC LAN completely shut down.

In the terminal window on your host laptop, start a new target PC LAN by issuing the **runlan** command. Arrange the target PC windows as you have done in the previous labs.

Start terminal windows on both machine1 and machine2, and become root on both target PCs by using the “sudo su –“ command.

In this lab we are going to use the iptables and tc (traffic control) tools to configure the Linux networking stack to control the rate of network traffic on eth0.

Go to machine1 and enter the following commands.

```
# ip addr add 192.168.10.10/24 dev eth0
# ip link set dev eth0 up
# iperf -s
```

Now do the following on machine2.

```
# ip addr add 192.168.10.11/24 dev eth0
# ip link set dev eth0 up
# iperf –c 192.168.10.10 –i 1 –t 10000
```

At this point, the iperf throughput is running at top speed, and will appear similar to the output below.

```
    [  3] local 192.168.10.11 port 37831 connected with 192.168.10.10 port 5001
    [ ID] Interval       Transfer     Bandwidth
    [  3]  0.0- 1.0 sec  42.4 MBytes   355 Mbits/sec
    [  3]  1.0- 2.0 sec  36.9 MBytes   309 Mbits/sec
    [  3]  2.0- 3.0 sec  44.9 MBytes   376 Mbits/sec
    [  3]  3.0- 4.0 sec  41.9 MBytes   351 Mbits/sec
    [  3]  4.0- 5.0 sec  36.9 MBytes   309 Mbits/sec
    [  3]  5.0- 6.0 sec  38.5 MBytes   323 Mbits/sec
    [  3]  6.0- 7.0 sec  38.8 MBytes   325 Mbits/sec
    [  3]  7.0- 8.0 sec  39.1 MBytes   328 Mbits/sec
    [  3]  8.0- 9.0 sec  37.9 MBytes   318 Mbits/sec
```

Start a second terminal on machine2, and become root. Perform the following configuration commands in this new terminal on **machine2**.

```
   iptables -F -t mangle
   iptables -A OUTPUT -t mangle -p tcp --dport 5001 -j MARK --set-mark 10

   tc qdisc add dev eth0 root handle 1:0 htb
   tc class add dev eth0 parent 1:0 classid 1:10 htb rate 5000kbit ceil 6000kbit prio 0
   tc filter add dev eth0 parent 1:0 prio 0 protocol ip handle 10 fw flowid 1:10
```

In this command sequence, we started by using iptables to flush (clear) the mangle table on machine2. Then we added a rule to the mangle table on the OUTPUT chain. In this rule, we set a mark of 10 on any TCP packet going to destination port 5001.

Next, we used the `tc` (traffic control) tool to add a new root qdisc (queue discipline) for eth0. This qdisc has handle “1:0”, and uses Hierarchy Token Bucket (htb).

We added a class, with classid 1:10 and attached it to the qdisc 1:0. In this class we use rate limiting to set the rate of this traffic to 5Mbit/sec, with a maximum ceiling rate of 6Mbit/sec.

Finally we added a filter to send packets marked with the mark of 10 to class 1:10.

This configuration sends our iperf output traffic (destination port 5001) through traffic shaping logic. Once the filter above is added, you will see the traffic rate change as shown here.

```
    [  3] 1256.0-1257.0 sec   512 KBytes  4.19 Mbits/sec
    [  3] 1257.0-1258.0 sec   512 KBytes  4.19 Mbits/sec
    [  3] 1258.0-1259.0 sec   512 KBytes  4.19 Mbits/sec
    [  3] 1259.0-1260.0 sec   640 KBytes  5.24 Mbits/sec
    [  3] 1260.0-1261.0 sec   512 KBytes  4.19 Mbits/sec
```

As you can see, the traffic control logic in the Linux kernel is able to manage the network traffic rate effectively. This can be used in more complex network setups to prevent some applications from using more than their fair share of network bandwidth.

Exit all terminals on the target PCs. Shut down the target PCs completely.

Lab Completed.

Lab 7. Firewall

Objective

Implement a firewall using iptables. Then, investigate a GUI firewall builder.

Procedure

Start with the target PC LAN completely shut down.

In the terminal window on your host laptop, start a new target PC LAN by issuing the **runlan** command. Arrange the target PC windows as you have done in the previous labs.

Start terminal windows on both machine1 and machine2, and become root on both target PCs by using the "sudo su –" command.

Go to machine1 and enter the following command.

```
# ip addr add 192.168.10.10/24 dev eth0
# ip link set dev eth0 up
```

Now do the following on machine2.

```
# ip addr add 192.168.10.11/24 dev eth0
# ip link set dev eth0 up
```

On machine1, connect to machine2 as student via ssh.

```
# ssh student@192.168.10.11
```

You will be asked for a confirmation – type "yes". Then enter the student password to log in. The important thing to learn here is that machine2 is open to outside connections and is accepting ssh logins.

Exit the ssh session now to get back to machine1.

Go to the terminal on machine2, and enter the following sequence of iptables commands to build up a simple but functional firewall.

First, we will flush all of the iptables chains to start with a fresh set of rules.

```
# iptables –F
```

Then we set the default policies for each predefined chain. These policies will only affect packets that are not matched by the rules that follow below.

```
iptables -P INPUT ACCEPT
iptables -P OUTPUT ACCEPT
iptables -P FORWARD DROP
```

The next rule allows input traffic that is part of an established connection. That includes traffic from connections that we initiated, but it does not include incoming connection requests.

```
iptables -A INPUT -m state --state RELATED,ESTABLISHED -j ACCEPT
```

All other input packets from the eth interfaces are dropped.

```
iptables -A INPUT -i eth+ -p udp -j DROP
iptables -A INPUT -i eth+ -p tcp -m tcp --syn -j DROP
```

Finally, we accept all traffic on the loopback interface for services within our local host.

```
iptables -A INPUT -i lo -j ACCEPT
```

Now go to machine1 and try the ssh login as student again.

```
# ssh student@192.168.10.11
```

The incoming connection request is being dropped by your firewall on machine2, so the connection request times out after a short while. Wait for it to time out. The firewall is protecting machine2 from incoming connection attempts. It is still capable of making outgoing connections, but it is not accepting them from other machines.

Let's look at a firewall with a GUI interface. First, go to machine2 and clear out the firewall rules that we just entered.

On machine2:

```
# iptables –F
```

Now on machine2, start the ufw firewall.

```
# ufw enable
```

Attempt to ssh from machine1 to machine2 as student.

```
# ssh student@192.168.10.11
```

The ufw firewall prevents the connection attempt. On machine2, take a look at the default iptables rules that were put in place when you enabled ufw. There are quite a few default rules in this firewall, so pipe the output to “less”.

```
# iptables –v --list | less
```

Let’s say that on machine2 we decide that we want to add a rule at the command line to allow incoming ssh connections. To do that, we use the following command.

```
# ufw allow 22/tcp
```

That says to allow all TCP traffic on port 22. Go to machine1 and try to ssh to machine2 as student.

```
# ssh student@192.168.10.11
```

The connection will be accepted. Be sure to exit the ssh session to get back to the machine1 prompt.

Now go to machine2 and enter the following command.

```
# gufw
```

This is the GUI interface to ufw. At the lower right of the gufw window, click the unlock icon so you can configure the firewall. You will see your custom rules regarding the ssh (port 22) connections on the screen. One rule was created for IPv4, and one for IPv6. Highlight the rules one at a time and remove them both by choosing Edit -> Remove Rule.

Return to machine1 and try the same ssh connection as before.

```
# ssh student@192.168.10.11
```

As expected, the connection attempt will just time out, since the machine2 firewall is again not allowing it.

Go back to machine2, and in gufw, let’s add a rule using the GUI. Click on Edit -> Add Rule. Click the “Preconfigured” tab. Using the pulldown menu choices, create a rule that reads “Allow In Service SSH”. Then click Add. Finally, click Close.

Now it is time for one last ssh attempt from machine1 to machine2.

```
# ssh student@192.168.10.11
```

The connection will succeed.

We have seen several different ways to create, modify, and display iptables firewall rules, both using the command line and using the gufw GUI interface. This will give you a good foundation for building your own firewalls in the future.

Exit all terminals on the target PCs. Shut down the target PCs completely.

Lab Completed.

Lab 8. Monitoring Network Traffic

Objective

Monitor network connections and traffic using tshark and wireshark.

Procedure

Start with the target PC LAN completely shut down.

In the terminal window on your host laptop, start a new target PC LAN by issuing the **runlan** command. Arrange the target PC windows as you have done in the previous labs.

Start terminal windows on both machine1 and machine2, and become root on both target PCs by using the "sudo su –" command.

Go to machine1 and enter the following command.

```
# ip addr add 192.168.10.10/24 dev eth0
# ip link set dev eth0 up
```

Now do the following on machine2.

```
# ip addr add 192.168.10.11/24 dev eth0
# ip link set dev eth0 up
# /etc/init.d/openbsd-inetd start
# tshark –S –f "tcp" –w out1.cap
```

You can safely ignore the error/warning message that occurs due to running tshark as root in this instance.

We configured eth0 to have an IP address, then we started the openbsd-inetd service, which will allow us to accept incoming telnet connections. We started tshark to monitor TCP traffic and capture it to a file, out1.cap.

Go to machine1 and use **telnet** to connect to machine2.

```
# telnet 192.168.10.11
```

Login as user "student" with password "student". As soon as you are logged in, type "exit" to end the telnet session.

Go back to machine2, and use <Ctrl>-C to stop the tshark capture. We now have the telnet connection traffic captured in file out1.cap. On machine2, use wireshark to view the captured packets.

```
# wireshark out1.cap
```

Click OK to get through the warnings that occur due to running wireshark as root. It is perfectly safe in this instance.

Click on Analyze -> Follow TCP Stream. In the window that appears, look at what was found in the TCP network traffic. You will notice that the color of the text indicates the direction that the data flowed. For the login name, each character in “student” appears twice (once in each color) because the characters were echoed back from the server as they were typed. The password was not echoed, so we see it clearly in plaintext here.

The lesson to be learned is that telnet is insecure, and your password can easily be seen on the network if you use telnet.

Let’s compare this with ssh session traffic. Go to machine2 and completely close wireshark. Then start a new tshark capture.

```
# tshark –S –f "tcp" –w out2.cap
```

Again, ignore the warning message.

On machine1, connect to machine2 as student using ssh.

```
# ssh student@192.168.10.11
```

Login all the way, then type “exit” to end the ssh session.

Go back to machine2 and hit <Ctrl>-C to stop the tshark capture. View the capture file using wireshark. Again, ignore the warning messages.

```
# wireshark out2.cap
```

Click on Analyze -> Follow TCP Stream. Notice that there is an exchange of encryption information, but after that there is no readable data in the data stream. It is completely secured. The username and password are not able to be detected in the network traffic by tshark/wireshark.

Close wireshark.

Exit all terminals on the target PCs. Shut down the target PCs completely.

Lab Completed.

Lab 9. Service Discovery Daemon

Objectives

Configure the Avahi service discovery daemon.

Procedure

Start with the target PC LAN completely shut down.

In the terminal window on your host laptop, start a new target PC LAN by issuing the **runlan** command. Arrange the target PC windows as you have done in the previous labs.

Start terminal windows on both machine1 and machine2, and become root on both target PCs by using the “sudo su –“ command.

Go to machine1 and enter the following commands.

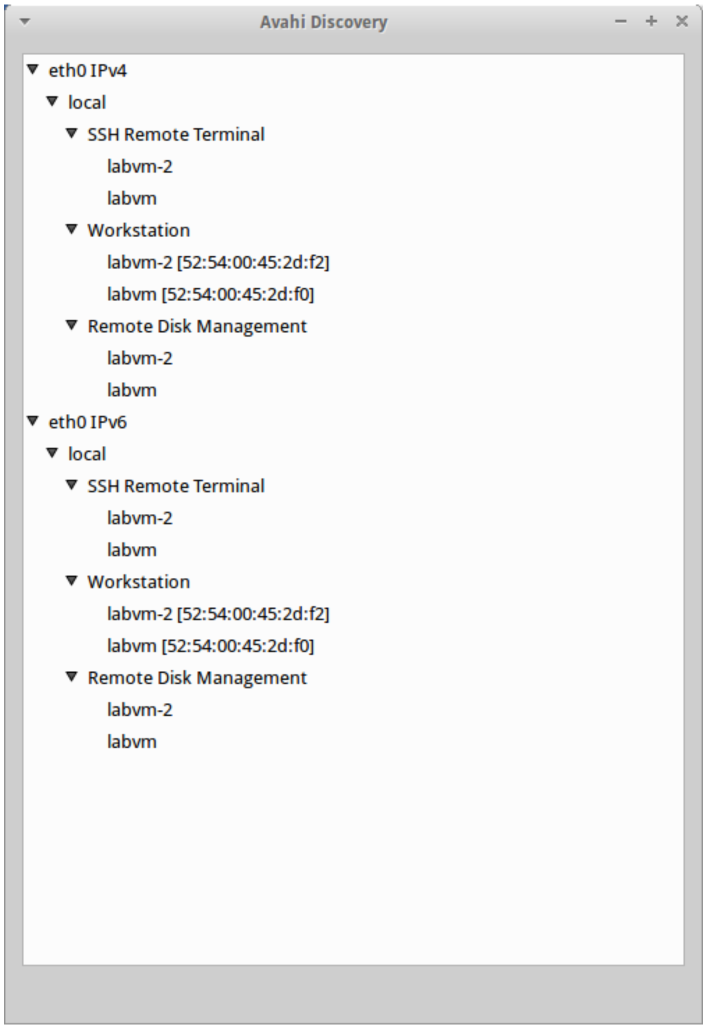
```
# ip addr add 192.168.10.10/24 dev eth0
# ip link set dev eth0 up
# avahi-discover
```

We gave eth0 an IP address and then started the interface that will show us what the Avahi service discovery daemon has discovered. Drag this Avahi display to the left, and expand it vertically so we can see as much as possible in the Avahi window. We want to be able to continue to see this window as we start typing on machine2. Notice how the number of discovered services changes once we configure the IP address on machine2.

Do the following on machine2.

```
# ip addr add 192.168.10.11/24 dev eth0
# ip link set dev eth0 up
```

After a few seconds, we see the Avahi window on machine1 has discovered several things about machine2. The Avahi daemon on machine2 is advertising certain services. Note that in the GUI display the name labvm-2 is used as a differentiator since both target PCs use the hostname labvm.



```
▼ eth0 IPv4
  ▼ local
    ▼ SSH Remote Terminal
        labvm-2
        labvm
    ▼ Workstation
        labvm-2 [52:54:00:45:2d:f2]
        labvm [52:54:00:45:2d:f0]
    ▼ Remote Disk Management
        labvm-2
        labvm
▼ eth0 IPv6
  ▼ local
    ▼ SSH Remote Terminal
        labvm-2
        labvm
    ▼ Workstation
        labvm-2 [52:54:00:45:2d:f2]
        labvm [52:54:00:45:2d:f0]
    ▼ Remote Disk Management
        labvm-2
        labvm
```

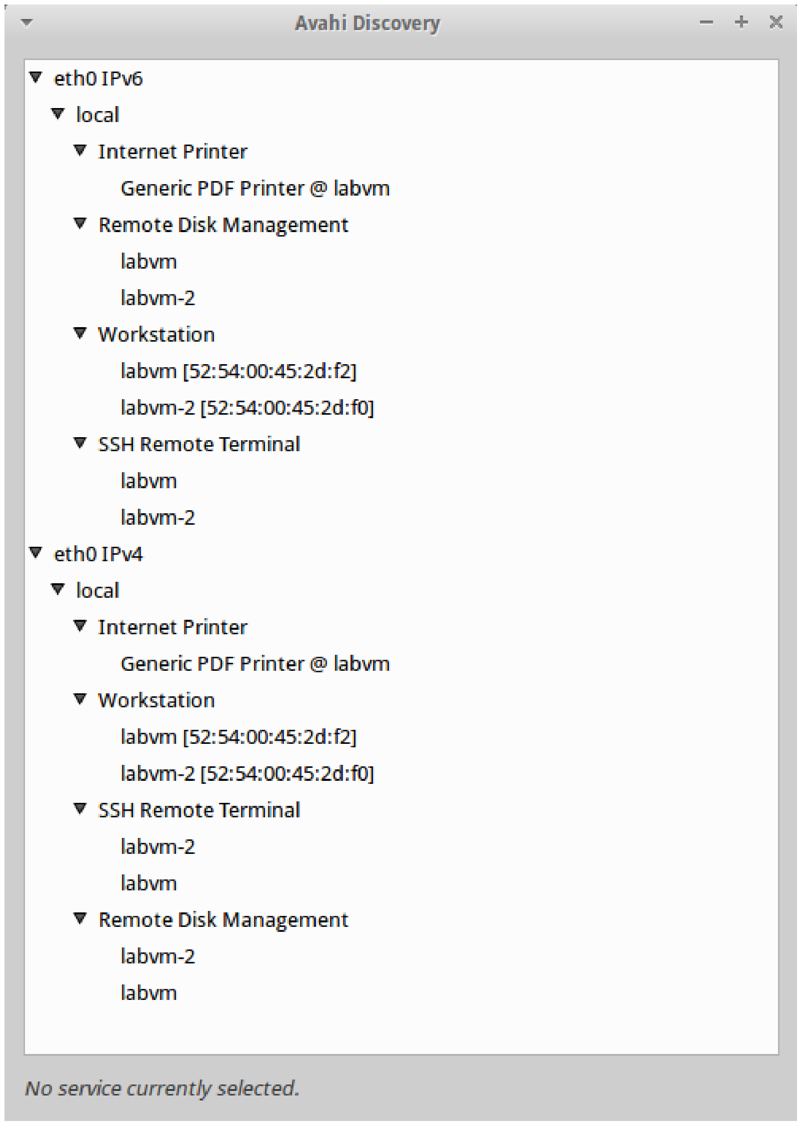
Go to machine2 and right-click on its desktop background. In the menu that pops up, select Applications -> Internet -> Firefox Web Browser. In the Firefox browser's URL window, delete the default URL that is there, and replace it with the following: "localhost:631". Then hit enter. Be sure to enter the colon character ":" before the "631". This connects to port 631 on the localhost (machine2). This is the CUPS printer system configuration page.

In the CUPS page that appears, choose the Printers tab. Here you will see that the system has a printer defined, called "Generic-PDF".

At the top of this web page, choose the Administration tab. On the Administration screen, under "Server Settings", check the box for "Share printers connected to this system". Leave other boxes unchecked.

Click the "Change Settings" button.

In the "Authentication Required" box, enter User Name "student" and Password "student". Now as you are about to hit the "OK" button, be sure that you are watching the Avahi display on machine1 carefully. Within a few seconds, your shared printer is discovered on machine1 by Avahi.



```
▼ eth0 IPv6
  ▼ local
    ▼ Internet Printer
        Generic PDF Printer @ labvm
    ▼ Remote Disk Management
        labvm
        labvm-2
    ▼ Workstation
        labvm [52:54:00:45:2d:f2]
        labvm-2 [52:54:00:45:2d:f0]
    ▼ SSH Remote Terminal
        labvm
        labvm-2
▼ eth0 IPv4
  ▼ local
    ▼ Internet Printer
        Generic PDF Printer @ labvm
    ▼ Workstation
        labvm [52:54:00:45:2d:f2]
        labvm-2 [52:54:00:45:2d:f0]
    ▼ SSH Remote Terminal
        labvm-2
        labvm
    ▼ Remote Disk Management
        labvm-2
        labvm
```

No service currently selected.

On machine2 you can just close the window that asks about remembering passwords. Now close the Firefox browser and the Avahi GUI.

Exit all terminals on the target PCs. Shut down the target PCs completely.

Lab Completed.

Lab 10. VPN

Objective

Configure and bring up a virtual private network using OpenVPN. Test the speed of the VPN versus an unencrypted link.

Procedure

Start with the target PC LAN completely shut down.

In the terminal window on your host laptop, start a new target PC LAN by issuing the **runlan** command. Arrange the target PC windows as you have done in the previous labs.

Start terminal windows on both machine1 and machine2, and become root on both target PCs by using the "sudo su –" command.

Go to machine2 and do the following.

```
# ip addr add 192.168.10.11/24 dev eth0
# ip link set dev eth0 up
```

Now go to machine1 and run these commands.

```
# ip addr add 192.168.10.10/24 dev eth0
# ip link set dev eth0 up
# cd /etc/openvpn
# openvpn --genkey --secret static.key
# scp static.key student@192.168.10.11:
```

On machine1 we gave eth0 an IP address, then we change to the **`/etc/openvpn/`** directory and create a static secret key. Then we used scp to copy that key to machine2. Remember to use the colon ":" at the end of the last command above, or it will not work. The scp command will prompt you to continue – type "yes". Then enter the student password "student".

The scp command above copied the static.key file to machine2, and it is now in the /home/student directory on machine2. Go to machine2 now to move it to the /etc/openvpn/ directory.

On machine2.

```
# cd /etc/openvpn
# mv /home/student/static.key .
```

Now still on machine2 in the /etc/openvpn/ directory, create new file lab.conf.

```
# vi lab.conf
```

Put the following 3 exact lines in file lab.conf.

```
dev tun
ifconfig 10.0.0.11 10.0.0.10
secret static.key
```

Now make sure you are still in the /etc/openvpn/ directory, and start the openvpn server.

```
# openvpn lab.conf
```

Go to machine1. On machine1, make sure that you are in the /etc/openvpn/ directory. Create file lab.conf.

```
# cd /etc/openvpn
# vi lab.conf
```

Put the following 4 exact lines in file lab.conf on machine1.

```
remote 192.168.10.11
dev tun
ifconfig 10.0.0.10 10.0.0.11
secret static.key
```

Now start the openvpn client on machine1. Be sure you are in the /etc/openvpn/ directory.

```
# openvpn lab.conf
```

Watch the screens on both machines. There is an initialization sequence that occurs, that takes about 10 seconds. At the end of this 10 second sequence, on both machines, you should see a message on the terminal running openvpn that says “Initialization Sequence Completed”.

On both machines, start a new terminal, type “ip addr show”, and look at the interface called “tun0”. This is the VPN interface.

Now, let’s test the speed of the VPN relative to an unencrypted interface.

Go to machine2 and run the iperf server as follows. This command can be run as root or as the student user.

```
  $ iperf -s
```

Next, go to machine1 and run the iperf client on the VPN interface as follows.

```
$ iperf -c 10.0.0.11 -i 1 -t 15
------------------------------------------------------------
Client connecting to 10.0.0.11, TCP port 5001
TCP window size: 20.1 KByte (default)
------------------------------------------------------------
[  3] local 10.0.0.10 port 46389 connected with 10.0.0.11 port 5001
[ ID] Interval       Transfer     Bandwidth
[  3]  0.0- 1.0 sec  9.75 MBytes  81.8 Mbits/sec
[  3]  1.0- 2.0 sec  10.0 MBytes  83.9 Mbits/sec
[  3]  2.0- 3.0 sec  11.8 MBytes  98.6 Mbits/sec
[  3]  3.0- 4.0 sec  9.88 MBytes  82.8 Mbits/sec
[  3]  4.0- 5.0 sec  10.9 MBytes  91.2 Mbits/sec
[  3]  5.0- 6.0 sec  12.9 MBytes   108 Mbits/sec
[  3]  6.0- 7.0 sec  13.0 MBytes   109 Mbits/sec
[  3]  7.0- 8.0 sec  14.8 MBytes   124 Mbits/sec
[  3]  8.0- 9.0 sec  14.9 MBytes   125 Mbits/sec
[  3]  9.0-10.0 sec  14.8 MBytes   124 Mbits/sec
...
```

Note that since these virtual interfaces are running at rates bound only by the CPU, your numbers may vary based on what else your system is doing at the time.

Next run the iperf client over the unencrypted eth0 interface.

```
$ iperf -c 192.168.10.11 -i 1 -t 15
------------------------------------------------------------
Client connecting to 192.168.10.11, TCP port 5001
TCP window size: 21.0 KByte (default)
------------------------------------------------------------
[  3] local 192.168.10.10 port 48008 connected with 192.168.10.11 port 5001
[ ID] Interval       Transfer     Bandwidth
[  3]  0.0- 1.0 sec  42.5 MBytes   357 Mbits/sec
[  3]  1.0- 2.0 sec  42.6 MBytes   358 Mbits/sec
[  3]  2.0- 3.0 sec  38.8 MBytes   325 Mbits/sec
[  3]  3.0- 4.0 sec  36.1 MBytes   303 Mbits/sec
[  3]  4.0- 5.0 sec  36.5 MBytes   306 Mbits/sec
[  3]  5.0- 6.0 sec  41.8 MBytes   350 Mbits/sec
[  3]  6.0- 7.0 sec  42.5 MBytes   357 Mbits/sec
[  3]  7.0- 8.0 sec  47.0 MBytes   394 Mbits/sec
[  3]  8.0- 9.0 sec  51.4 MBytes   431 Mbits/sec
[  3]  9.0-10.0 sec  40.1 MBytes   337 Mbits/sec
...
```

The encryption of the network traffic on the VPN does affect throughput. This is offset however by the security benefits of using a VPN.

Exit all terminals on the target PCs. Shut down the target PCs completely.

Lab Completed.