Bridging Network Connections

Contents

Introduction

Bridging your network connection is a handy method for sharing your internet connection between two (or more) computers. It’s useful if you can’t buy a router with more than one ethernet port, or if you’re a college student in a dorm room with limited ethernet jacks and no router.

Basically, bridging is plugging one computer into another computer that already has a connection to a larger network (like the internet) and letting the bridged computer use the networked computer’s connection. To do so though, the networked computer needs to have two ethernet ports, one for the big network, and one for the bridged computer. Make sure that before starting that the computer you’re gonna

bridge through has two ethernet ports, and that the hardware is capable of bridging ethernet connections (it probably should be).

Another example scenario for using bridging is to provide redundant networking capabilities. For example using two network interfaces to connect to two spanning tree enabled switches provides a redundant connection in the event of a cable, interface or switch failure. This requires spanning tree to be enabled on both the bridge interface and the switch.

Installing the software

The program you're going to need is called `brctl` and is included in DebianPkg: bridge-utils. Find it in Synaptic, or install it using this command:

```
# aptitude install bridge-utils
```

This program will allow us to set up and use the bridge interface. The bridge interface appears as a new interface in `ip link`, much like `eth0` or `eth1`. It doesn't physically exist on your computer, but instead it is a virtual interface that just takes the packets from one physical interface, and transparently routes them to the other.

Setting up your Bridge

Manual bridge setup

Note: All these commands are to be issued on the computer with the existing network connection. To set up the computer that's going to be bridged, just set it up normally, as you would any other computer. You CAN use DHCP, or you can use a static address. It doesn't matter.

Note: If, after trying to use the bridge interface, you find your network link becomes dead and refuses to work again, it might be that the router/switch upstream is blocking "unauthorized switches" in the network (for example, by detecting BPDU packets). You'll have to change its configuration to explicitly allow the host machine/network port as a "switch".

First step to creating the bridge network is actually creating it. Issue this command to get the ball rolling and create the new interface.

```
# brctl addbr br0
```

The name `br0` is totally up to you, this is just an example name that I've chosen for the wiki article. Anyway, now that you have your bridge device, you have to add the interfaces that are gonna be bridged. You can cross-check the enumeration of your ethernet devices with (`eth0`, `eth1`, etc. is common):

```
# ip addr show
```

Add both the interface with the second computer, and the interface that leads to the existing network. Do it with this command:

```
# brctl addif br0 eth0 eth1
```

This will add the two interfaces `eth0` and `eth1` to bridge `br0`. Simple enough. There's no distinction with how you add the bridges, or what order you do it, or any special commands

you have to add to distinguish them. So don't worry about that.

Well, now we have our bridges, so bring all the interfaces up, and you'll be set!

Configuring bridging in /etc/network/interfaces

To make your bridge a little more permanent, you're gonna need to edit `/etc/network/interfaces`. Using our example names, make it look like this and you're set (if you want to use DHCP):

```
# This file describes the network interfaces available
# and how to activate them. For more information, see

# The loopback network interface
auto lo br0
iface lo inet loopback

# Set up interfaces manually, avoiding conflicts with,
iface eth0 inet manual

iface eth1 inet manual

# Bridge setup
iface br0 inet dhcp
       bridge_ports eth0 eth1
```

To bring up your bridge, you just have to issue `# ifup br0` and it'll bring up the other necessary interfaces without anything in your `interfaces` file about the bridged interfaces.

If you like static IP's, then you can just add the static IP options under the `br0` interface setup. Kinda like this:

```
# This file describes the network interfaces available
# and how to activate them. For more information, see

# The loopback network interface
auto lo br0
iface lo inet loopback

# Set up interfaces manually, avoiding conflicts with,
iface eth0 inet manual

iface eth1 inet manual

# Bridge setup
iface br0 inet static
       bridge_ports eth0 eth1
       address 192.168.1.2
       broadcast 192.168.1.255
       netmask 255.255.255.0
       gateway 192.168.1.1
```

Useful options for virtualised environments

Some other useful options to use in any stanza in a virtualised environment are:

```
       bridge_stp off       # disable Spanning Tree Pr
       bridge_waitport 0    # no delay before a port b
       bridge_fd 0          # no forwarding delay
```

```
        bridge_ports none    # if you do not want to bi
        bridge_ports regex eth* # use a regular express
```

Setting up bridge-related kernel variables

There are several kernel variables that affect bridge operation. In some cases you may need to tweak these variables. There are two common options:

- Add variables to */etc/sysctl.conf* directly
- Put them to a sysctl configuration file fragment (e.g. */etc/sysctl.d/bridge_local.conf*)

In the latter case, the *procps* init script *should* take care of loading them during boot. However, on Squeeze it does not, and you need to restart it from */etc/rc.local* (or similar):

```
# /etc/rc.local

# Load kernel variables from /etc/sysctl.d
/etc/init.d/procps restart

exit 0
```

Libvirt and bridging

Libvirt is a virtualization API which supports KVM and various other virtualization techniques. In many cases, it's desirable to share a physical network interface with guests, i.e. setup a bridge they can use. This operation is composed of two parts:

- Setup the bridge interface on host as described in this

article and in here

- Configure guest to use the newly-created bridge

You can verify if bridging is working properly by looking at *brctl* output:

```
root@server:/etc/libvirt/qemu# brctl show
bridge name     bridge id               STP enabled
br0             8000.001ec952d26b       yes




virbr0          8000.000000000000       yes
```

As can be seen, guest network interfaces *vnet0*, *vnet1* and *vnet2* are bound with the physical interface *eth0* in the bridge *br0*. The *virbr0* interface only used for NAT connectivity with libvirt.

Bridging with a wireless NIC

Just like you can bridge two wired ethernet interfaces, you can bridge between an ethernet interface and a wireless interface. However, most Access Points (APs) will reject frames that have a source address that didn't authenticate with the AP. Since Linux does ethernet bridging transparently (doesn't modify outgoing or incoming frames), we have to set up some rules to do this with a program called DebianPkg: ebtables.

ebtables Overview

ebtables is essentially like DebianPkg: iptables, except it operates on the MAC sublayer of the data-link layer of the

OSI model, instead of the network layer. In our case, this allows to change the source MAC address of all of our frames. This is handy because we fool our AP into thinking that all of our forwarded frames come from the machine which authenticated to the AP.

bridge-utils Modifications

Before this will work, you need to modify your `/etc/network/interfaces` file, and add this line to your bridge stanza:

```
pre-up iwconfig wlan0 essid $YOUR_ESSID
bridge_hw $MAC_ADDRESS_OF_YOUR_WIRELESS_CARD
```

Obviously replacing `$MAC_ADDRESS_OF_YOUR_WIRELESS_CARD` with the actual MAC address of your wireless card, and `$YOUR_ESSID` as the ESSID of your wireless network. If you don't know your MAC address, you can find it by typing

```
# ip link show wlan0
```

Where `wlan0` is your wireless interface. Your MAC address is listed as the HWaddr.

Setting up the rules

First, install ebtables:

```
# aptitude install ebtables
```

Now we can start setting up the rules. The syntax for ebtables is almost identical to that of iptables, so if you have experience with iptables, this will look pretty familiar to you.

The first rule we're going to set up will change the source MAC address of all our incoming frames from the AP to the MAC address of our bridge.

```
# ebtables -t nat -A POSTROUTING -o wlan0 -j snat --to
```

The next rules require you to know the MAC and IP of each of the machines behind your bridge. Replace $MAC and $IP with these.

```
# ebtables -t nat -A PREROUTING -p IPv4 -i wlan0 --ip-
# ebtables -t nat -A PREROUTING -p ARP -i wlan0 --arp-
```

This is tedious to have to type in everytime you add a new computer to a switch behind your bridge, so I wrote a script to do it for you

Toggle line numbers

```
1  #!/bin/bash
2  # addcomputer
3  # Will Orr - 2009
4
5  INIF="wlan0"
6
7  function add_ebtables () {
8    COMPIP=$1
9    COMPMAC=$2
```

```
10
11    ebtables -t nat -A PREROUTING -i $INIF -p IF
12    dnat --to-dst $COMPMAC --dnat-target ACCEPT
13    ebtables -t nat -A PREROUTING -i $INIF -p AF
14    -j dnat --to-dst $COMPMAC --dnat-target ACCE
15
16  }
17
18  if [[ $# -ne 2 ]]; then
19    echo "Usage: $0 ip mac"
20  elif [[ $(whoami) != "root" ]]; then
21    echo "Error: must be root"
22  else
23    add_ebtables $1 $2
24  fi
```

Saving your rules

After you have written your ebtables rules, you need to save them in an atomic file. Otherwise, your rules will not be preserved. Saving them is rather simple though.

```
# EBTABLES_ATOMIC_FILE=/root/ebtables-atomic ebtables
```

And then load them like this:

```
# EBTABLES_ATOMIC_FILE=/root/ebtables-atomic ebtables
```

If you want to load your ebtables rules at boot time, a handy place to stick the commit command is in /etc/rc.local. Just

pop it in there before the `exit 0` line.

More information

- http://www.linux.com/feature/133849

CategoryNetwork