



[Translation\(s\):](#) English - [Français](#)

Bridging Network Connections

Contents

1. [Bridging Network Connections](#)

1. [Introduction](#)

2. [Installing the software](#)

3. [Setting up your Bridge](#)

1. [Manual bridge setup](#)

2. [Configuring bridging in /etc/network/interfaces](#)

1. [Useful options for virtualised environments](#)

3. [Setting up bridge-related kernel variables](#)

4. [Libvirt and bridging](#)

5. [Bridging with a wireless NIC](#)

1. [ebtables Overview](#)

2. [bridge-utils Modifications](#)

3. [Setting up the rules](#)

4. [Saving your rules](#)

6. [Link Aggregation \(LACP\) with VLANs](#)

7. [More information](#)

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 before starting that the computer you're going to 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 software allows you 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 up to you and can be anything you want. Now you need to add the interfaces that are going to 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. Use `brctl`:

```
# 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 will 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 on your system
# and how to activate them. For more information, see interfaces(5).

# The loopback network interface
auto lo
iface lo inet loopback

# Set up interfaces manually, avoiding conflicts with, e.g., network manager
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. For example:

```
# This file describes the network interfaces available on your system
# and how to activate them. For more information, see interfaces(5).
```

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

# Set up interfaces manually, avoiding conflicts with, e.g., network manager
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 Protocol
bridge_waitport 0    # no delay before a port becomes available
bridge_fd 0          # no forwarding delay
bridge_ports none    # if you do not want to bind to any ports
bridge_ports regex eth* # use a regular expression to define ports
```

Setting up bridge-related kernel variables

There are several kernel variables that affect bridge operation. In [!\[\]\(eafc244b53721dd1ec133f0772f70fc7_img.jpg\) 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 `procp`s 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

[!\[\]\(e1d6102fe77919492c04879c8450f1f5_img.jpg\) Libvirt](#) is a virtualization API that supports KVM (and various other virtualization technologies). It's often desirable to [!\[\]\(f18214e08965a1644d0b2b0878fd365f_img.jpg\) share a physical network interface with guests](#) by creating a bridge. This usually offers excellent performance and doesn't require NAT. This operation is composed of two parts:

- Setup the bridge interface on host as described in this article, [!\[\]\(c33cb967c8fc4f5e27188a389b621c8e_img.jpg\) here](#) or [!\[\]\(38e1383487ca0f0e9e2c9378b9dbcae7_img.jpg\) here](#)
- Configure guest to use the newly-created bridge

The [!\[\]\(d5d7044e5caf6907399af2dced8d6ff8_img.jpg\) libvirt Networking Handbook](#) provides thorough instructions.

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    interfaces
br0               8000.001ec952d26b        yes            eth0
                  vnet0
                  vnet1
                  vnet2
virbr0            8000.00000000000000      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 is only used by libvirt to give guests NAT connectivity.

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](#).

For an alternative Layer 3 approach using proxy ARP and routing, see [BridgeNetworkConnectionsProxyArp](#).

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 set the source MAC address to the MAC address of the bridge for all frames sent to the AP.

```
# ebtables -t nat -A POSTROUTING -o wlan0 -j snat --to-src $MAC_OF_BRIDGE --snat-arp --snat-target ACC
```

The next rules will 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-dst $IP -j dnat --to-dst $MAC --dnat-target ACC  
# ebtables -t nat -A PREROUTING -p ARP -i wlan0 --arp-ip-dst $IP -j dnat --to-dst $MAC --dnat-target ACC
```

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 IPv4 --ip-dst $COMPIP -j \
__12   dn timer --to-dst $COMPMAC --dnat-target ACCEPT
__13   ebtables -t nat -A PREROUTING -i $INIF -p ARP --arp-ip-dst $COMPIP \
__14   -j dn timer --to-dst $COMPMAC --dnat-target ACCEPT
__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 -t nat --atomic-save
```

And then load them like this:

```
# EBTABLES_ATOMIC_FILE=/root/ebtables-atomic ebtables -t nat --atomic-commit
```

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.

Link Aggregation (LACP) with VLANs

Here is an example of the `/etc/network/interfaces` file for 2 interfaces LACP bonded together with VLANs defined on top of the bond.

Notes: Tested on Debian Jessie 8.0 rc1 on 3/11/2015 (AMD64 arch)

Required debian packages for vlan and bonding:

```
aptitude install vlan ifenslave  
echo "8021q" >> /etc/modules
```

The `/etc/network/interfaces` could be:

```
auto lo  
iface lo inet loopback  
  
# there is one other thing I am forgetting here... see bonding setup above  
  
##### Bonded 10Gig setup - LACP Aggregation ##### bond eth4 and eth5  
#### (set up from switch as trunk links with LACP) then split out VLANs
```

```
## View status with:  cat /proc/net/bonding/bond0
```

```
auto bond0
```

```
iface bond0 inet manual
```

```
    up ifconfig bond0 0.0.0.0 up
```

```
    slaves eth4 eth5
```

```
# bond-mode 4 = 802.3ad
```

```
    bond-mode 4
```

```
    bond-miimon 100
```

```
    bond-downdelay 200
```

```
    bond-updelay 200
```

```
    bond-lacp-rate 1
```

```
    bond-xmit-hash-policy layer2+3
```

```
auto vlan1023
```

```
iface vlan1023 inet static
```

```
    address 18.28.4.123
```

```
    netmask 255.255.255.0
```

```
    gateway 18.28.4.1
```

```
    vlan-raw-device bond0
```

```
auto vlan1024
```

```
iface vlan1024 inet static
```

```
    address 10.15.160.1
```

```
netmask 255.255.255.0  
gateway 10.15.160.100  
vlan-raw-device bond0
```

More information

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

[CategoryNetwork](#)