Linux BRIDGE−STP−HOWTO

Uwe Böhme

Johann−Heinrich−Abt−Straße 7
95213
Münchberg
Germany
+49/9251 960877
+49/9251 960878
uwe@bnhof.de

Lennert Buytenhenk

bridge code maintainer and developer
gnu.org

buytenh@gnu.org

Release v0.04

Revision History

Revision v0.04	11 January 2001	Revised by: U.B.
Changed Lennert`s Bridge Homepage URL; added NIC to list.		
Revision v0.03	17 July 2000	Revised by: U.B.
Overwork pdf. Download links in doc.		
Revision v0.02	16 July 2000	Revised by: U.B.
Fixed broken graphics in html dsl. Prepared pdf. Typos.		
Revision v0.01	25 June 2000	Revised by: U.B.
Changes name from BRIDGE−HOWTO to BRIDGE−STP−HOWTO (avoid interference with BRIDGE−HOWTO by Christopher Cole) and kill version 1.xx. Lennert Buytenhenk announced as coauthor.		
Revision v0.00	01 June 2000	Revised by: U.B.
Initial Release.		

Table of Contents

1. License

2. Document Home and Downloads

2.1. The Bridge Sources And Utilities

Official url is http://www.math.leidenuniv.nl/~buytenh/bridge/. With developer kernel 2.3.47 the new bridging code is part of the mainstream.

2.2. The Mailing−List

The Bridge−Mailinglist is homed at http://www.math.leidenuniv.nl/mailman/listinfo/bridge.

2.3. This Document

This document has it's official homepage at http://www.bnhof.de/~uwe/bridge−stp−howto/BRIDGE−STP−HOWTO/. It's a part of the Linux Documentation Project located at http://www.linuxdoc.org/.

Download Types and Locations

Build environment as tar.gziped file

http://www.bnhof.de/~uwe/bridge−stp−howto/BRIDGE−STP−HOWTO.tar.gz

HTML−gziped file

http://www.bnhof.de/~uwe/bridge−stp−howto/BRIDGE−STP−HOWTO.html.tar.gz

PDF−gziped file

http://www.bnhof.de/~uwe/bridge−stp−howto/BRIDGE−STP−HOWTO.pdf.gz

PS−gziped file

http://www.bnhof.de/~uwe/bridge−stp−howto/BRIDGE−STP−HOWTO.ps.gz

3. What Is A Bridge?

A bridge is a device that separates two or more network segments within one logical network (e.g. a single IP−subnet).

A bridge is usually placed between two separate groups of computers that talk with each other, but not that much with the computers in the other group. A good example of this is to consider a cluster of Macintoshes and a cluster of Unix machines. Both of these groups of machines tend to be quite chatty amongst themselves, and the traffic they produce on the network causes collisions for the other machines who are trying to speak to one another.

The job of the bridge is to examine the destination of the data packets one at a time and decide whether or not to pass the packets to the other side of the Ethernet segment. The result is a faster, quieter network with less collisions.

The bridging code decides whether to bridge data or to drop it not by looking at the protocol type (IP, IPX, NetBEUI), but by looking at the MAC−address unique to each NIC.

> **Important:** It's vital to understand that a bridge is neither a router nor a fire−wall. Spoken in simple term a bridge behaves like a network switch (i.e. Layer 2 Switch), making it a transparent network component (which is not absolutely true, but nearly). Read more about this at Section 4.

In addition, you can overcome hardware incompatibilities with a bridge, without leaving the address−range of your IP−net or subnet. E.g. it's possible to bridge between different physical media like 10 Base T and 100 Base TX.

My personal reason for starting to set up a bridge was that in my work I had to connect Fast Ethernet components to a existing HP Voice Grade network, which is a proprietary networking standard.

Features Above Pure Bridging

STP

> The Spanning Tree Protocol is a nifty method of keeping Ethernet devices connected in multiple paths working. The participating switches negotiate the shortest available path by STP. This feature will be discussed in Section 7.1.

Multiple Bridge Instances

> Multiple bridge instances allow you to have more than one bridge on your box up and running, and to control each instance separately.

Fire−walling

> There is a patch to the bridging code which allows you to use IP chains on the interface inside a bridge. More info about this you'll find at Section 7.2.

4. Rules On Bridging

There is a number of rules you are not allowed to break (otherwise your bridge will do).

- A port can only be a member of one bridge.
- A bridge knows nothing about routes.
- A bridge knows nothing about higher protocols than ARP. That's the reason why it can bridge any possible protocol possibly running on your Ethernet.
- No matter how many ports you have in your logical bridge, it's covered by only one logical interface
- As soon as a port (e.g. a NIC) is added to a bridge you have no more direct control about it.

Warning
If one of the points mentioned above is not clear to you now, don't continue reading. Read the documents listed in Appendix B first.

If you ever tried to ping an unmanaged switch, you will know that it doesn't work, because you don't have a IP−address for it. To switch datagrams it doesn't need one. The other thing is if you want to manage the switch. It's too much strain, to take a dumb terminal, walk to the place you installed it (normally a dark, dusty and warm room, with a lot of green and red Christmas lights), to connect the terminal and to change the settings.

What you want is remote management, usually by SNMP, telnet, rlogin or (best) ssh. For all this services you will need a IP. That's the exception to the transparency. The new code allows you without any problem to assign a IP address to the virtual interface formed by the bridge−instance you will create in Section 6.2. All NIC's (or other interfaces) in your bridge will happily listen and respond to datagrams destined to this IP.

All other data will not interfere with the bridge. The bridge just acts like a switch.

5. Preparing The Bridge

This section describes what you need and how you do to prepare your bridge.

5.1. Get The Files

Here you can find a list of the files and down−loads you will need for the setup of the bridge. If you have one of the mentioned files or packages on your distribution, of course there is no need to create network load.

I'll only mention the files for the 2.2.14 kernel. If you want to try a different one (e.g. 2.2.15 or the recent development kernel) just replace the kernel version number and look whether you find it.

> **Important:** You have read the *abstract*, didn't you? So you know that there is no need to download any kernel−patch if you're working with a kernel later than 2.3.47.

File and package list

Unpatched kernel−sources

E.g. `linux−2.2.14.tar.bz2` available from your local kernel.org mirror. Please check first if you find it in your distribution (take unpatched kernel−sources). If you don't, please check The Linux Kernel Archive Mirror System for a close by mirror and down−load it from there.

Bridge patches

> **Note:** If your kernel is later than 2.3.47 you don't need this. The bridging is part of the mainstream from that version.

Get the bridge kernel patches for your kernel version from http://www.math.leidenuniv.nl/~buytenh/bridge/. Identify the file by the kernel number.

> **Note:** There are also patches allowing to work with IP chains. I never tried it, for I don't see the need to fire−wall inside my LAN, and absolutely no need to bridge against the outer world. Feel free to contribute about that issue.

Kernel patches for the stable 2.2 kernel.

Available Kernel patches

bridge−0.0.9−against−2.2.18.diff, the main kernel patch against 2.2.18

http://www.math.leidenuniv.nl/~buytenh/bridge/patches/bridge−0.0.9−against−2.2.18.diff

bridge−ipchains−against−0.0.9−against−2.2.18.diff, an add−on patch for bridge firewalling against 2.2.18

http://www.math.leidenuniv.nl/~buytenh/bridge/patches/bridge−ipchains−against−0.0.9−against−2.2.18.diff *bridge−0.0.8−against−2.2.18pre19.diff, the main kernel patch against 2.2.18pre19.*

http://www.math.leidenuniv.nl/~buytenh/bridge/patches/bridge−0.0.8−against−2.2.18pre19.diff

bridge−0.0.8−against−2.2.17−0.5.diff, the main kernel patch against 2.2.17−0.5

http://www.math.leidenuniv.nl/~buytenh/bridge/patches/bridge−0.0.8−against−2.2.17−0.5.diff

bridge−ipchains−against−0.0.8−against−2.2.18pre19.diff, an add−on patch for bridge firewalling against 2.2.18pre19

http://www.math.leidenuniv.nl/~buytenh/bridge/patches/bridge−ipchains−against−0.0.8−against−2.2.18pre19.diff

bridge−ipchains−against−0.0.8−against−2.2.17−0.5.diff, an add−on patch for bridge firewalling against 2.2.17−0.5

http://www.math.leidenuniv.nl/~buytenh/bridge/patches/bridge−ipchains−against−0.0.8−against−2.2.17−0.5.diff

bridge−0.0.7−against−2.2.18pre15.diff, the main kernel patch against 2.2.18pre15

http://www.math.leidenuniv.nl/~buytenh/bridge/patches/bridge−0.0.7−against−2.2.18pre15.diff

bridge−ipchains−against−0.0.7−against−2.2.18pre15.diff, an add−on patch for bridge firewalling against 2.2.18pre15

http://www.math.leidenuniv.nl/~buytenh/bridge/patches/bridge−ipchains−against−0.0.7−against−2.2.18pre15.diff

bridge−0.0.7−against−2.2.17.diff, the main kernel patch against 2.2.17

http://www.math.leidenuniv.nl/~buytenh/bridge/patches/bridge−0.0.7−against−2.2.17.diff

bridge−ipchains−against−0.0.7−against−2.2.17.diff, an add−on patch for bridge firewalling against 2.2.17

http://www.math.leidenuniv.nl/~buytenh/bridge/patches/bridge−ipchains−against−0.0.7−against−2.2.17.diff

Bridge configuration utilities

You also will need the bridge configuration utilities to set up the bridge Section 6. You can also download them from http://www.math.leidenuniv.nl/~buytenh/bridge/.

5.2. Apply The Patches

Note: If your kernel is later than 2.3.47 you don't need this. The bridging is part of the mainstream from that version.

Apply the bridging patch your kernel. If you don`t know *how to* do that read the Kernel−HOWTO which can be found in your distribution or at http://www.linuxdoc.org/HOWTO/Kernel−HOWTO.html

Example 1. Applying a kernel patch

```
root@mbb-1:~ # cd /usr/src/linux-2.2.14
root@mbb-1:/usr/src/linux-2.2.14 # patch -p1 < \
    bridge-0.0.5-against-2.2.14.diff
.
.
```

5.3. Configure The Kernel

Now it's time we configure our freshly patched kernel to create the ability to bridge.

Run **make config**, **make menuconfig** or the click−o−rama **make xconfig**. Select **bridging** in the **networking option** section to be compiled as a module. AFAIK there is no strong reason why *not* to compile it as a kernel module, whereas I heard rumors about problems with compiling the bridging code directly into the kernel.

```
root@mbb-1:~ # cd /usr/src/linux-2.2.14
root@mbb-1:/usr/src/linux-2.2.14 # make menuconfig
.
```

5.4. Compile The Kernel

Compile your kernel Example 2. Make the new compiled kernel−image to be loaded. I don't know if the kernel patches only apply to the bridging−module or also modify some interfaces inside `vmlinuz`. So it might not be a error to give a reboot after you updated the kernel−image.

Example 2. Commands To Compile Your Kernel

```
root@mbb-1:/usr/src/linux-2.2.14 # make dep clean zImage modules modules_install zlilo
...
```

5.5. Compile The Bridge Utilities

This is how to compile and install from the scratch. Just **unzip** the utilities−tarball, **cd** into the newly created directory and give a **make**.

Example 3. Commands To Compile Your Bridge−Utilities

```
root@mbb-1:/usr/src/linux-2.2.14 # cd /usr/local/src
root@mbb-1:/usr/local/src/ # tar xzvf bridge-utils-0.9.1.tar.gz
.....
```

```
....
root@mbb-1:/usr/local/src # cd bridge
root@mbb-1:/usr/local/src/bridge # make
.....
....
```

After the compilation shown in Example 3 have worked properly, you can copy the executables to let's say `/usr/local/sbin/` (at least I did). So the commands you have to give should be clear, but to be complete see Example 4

Example 4. Copy The Binaries Of The Utilities

```
root@mbb-1:/usr/local/src/bridge # cd brctl
root@mbb-1:/usr/local/src/bridge/brctl # cp brctl /usr/local/sbin
root@mbb-1:/usr/local/src/bridge/brctl # chmod 700 /usr/local/sbin/brctl
root@mbb-1:/usr/local/src/bridge/brctl # cp brctld /usr/local/sbin
root@mbb-1:/usr/local/src/bridge/brctl # chmod 700 /usr/local/sbin/brctld
```

Also now you can copy the new man−page to a decent place, as shown in Example 5.

Example 5. Copy The Man−page Of brctl

```
root@mbb-1:/usr/local/src/bridge # cd doc
root@mbb-1:/usr/local/src/bridge/doc #  gzip -c brctl.8 > /usr/local/man/man8/brctl.8.gz
```

6. Set Up The Bridge

Make sure all your network cards are working nicely and are accessible. If so, **ifconfig** will show you the hardware layout of the network−interface. If you have problems making your cards work please read the Ethernet−HOWTO at http://www.linuxdoc.org/HOWTO/Ethernet−HOWTO.html. Don't mess around with IP−addresses or net−masks. You will not need it, until you bridge fully operational an up.

After you did the steps mentioned above a **modprobe −v bridge** should show no errors. You can check the success by issuing a **cat /proc/modules**. Also for each of the network cards you want to use in the bridge the **ifconfig** **`whateverNameYourInterfaceHas`** should give you some information about the interface.

If your bridge−utilities have been correctly built and your kernel and bridge−module are OK, then issuing a **brctl** should show a small command synopsis.

6.1. brctl Command Synopsis

```
root@mbb-1:~ # brctl
commands:
    addbr           <bridge>                add bridge                     ❶
    addif           <bridge> <device>       add interface to bridge        ❷
    delbr           <bridge>                delete bridge                  ❸
    delif           <bridge> <device>       delete interface from bridge   ❹
    show                                    show a list of bridges         ❺
    showbr          <bridge>                show bridge info               ❻
    showmacs        <bridge>                show a list of mac addrs       ❼

    setageing       <bridge> <time>         set ageing time                ❽
    setbridgeprio   <bridge> <prio>         set bridge priority            ❾
    setfd           <bridge> <time>         set bridge forward delay       ❿
    setgcint        <bridge> <time>         set garbage collection interval (11)
    sethello        <bridge> <time>         set hello time                 (12)
    setmaxage       <bridge> <time>         set max message age            (13)
    setpathcost     <bridge> <port> <cost>  set path cost                  (14)
    setportprio     <bridge> <port> <prio>  set port priority              (15)
    stp             <bridge> <state>        {dis,en}able stp               (16)
```

❶❸

The **brctl addbr `bridgename`** command creates a logical bridge instance with the name **`bridgename`**. You will need at least one logical instance to do any bridging at all. You can interpret the logical bridge being a container for the interfaces taking part in the bridging. Each bridging instance is represented by a new network interface.

Example 6. Creating A Instance

```
root@mbb-1:~ # brctl addbr mybridge1
```

The corresponding "shutdown" command is **brctl delbr `bridgename`**.

Caution

brctl delbr **`bridgename`** will only work, if there are no more interfaces added to the instance you want to delete.

❷❹

The **brctl addif** **`bridgename device`** command enslaves the network device **`device`** to take part in the bridging of **`bridgename`**. As a simple explanation, each interface enslaved into a instance is bridged to the other interfaces of the instance.

The corresponding command to tale a interface out of the bridge would be **brctl delif** **`bridgename device`**

❺

The **brctl show** command gives you a summary about the overall bridge status, and the instances running as shown in Example 7. If you are interested in detailed information about a instance and it's interfaces you will have to check ❻.

Example 7. Output Of brctl show

```
root@mbb-1:~ # brctl show
bridge name     bridge id               stp enabled
mybridge1       0000.0800062815f6       yes
```

❻

The **brctl showbr** **`bridgename`** command gives you a summary about a bridge instance and it's enslaved interfaces.

Example 8. Output Of brctl showbr `bridgename`

```
root@mbb-1:~ # brctl showbr mybridge1
mybridge1
 bridge id              0000.0800062815f6
 designated root        0000.0800062815f6
 root port                 0                    path cost                  0
 max age                   4.00                 bridge max age             4.00
 hello time                1.00                 bridge hello time          1.00
 forward delay             4.00                 bridge forward delay       4.00
 ageing time             300.00                 gc interval                4.00
 hello timer               0.84                 tcn timer                  0.00
 topology change timer     0.00                 gc timer                   1.84
 flags


eth0 (1)
 port id                8001                    state                   forwarding
 designated root        0000.0800062815f6       path cost                100
 designated bridge      0000.0800062815f6       message age timer          0.00
 designated port        8001                    forward delay timer        0.00
 designated cost           0                    hold timer                 0.84
 flags

eth1 (2)
 port id                8002                    state                   forwarding
 designated root        0000.0800062815f6       path cost                100
 designated bridge      0000.0800062815f6       message age timer          0.00
 designated port        8002                    forward delay timer        0.00
 designated cost           0                    hold timer                 0.84
 flags
```

❼

The **brctl showmacs `bridgename`** command gives you a list of MAC−addresses of the interfaces which are enslaved in **`bridgename`**.

Example 9. Output Of brctl showmacs `bridgename`

```
root@mbb-1:~ # brctl showmacs mybridge1
port no mac addr                is local?       ageing timer
  1     00:10:4b:b6:c6:e4       no               119.25
  1     00:50:04:43:82:85       no                 0.00
  1     00:50:da:45:45:b1       no                76.75
  1     00:a0:24:d0:4c:d6       yes                0.00
  1     00:a0:24:f0:22:71       no                 5.81
  1     08:00:09:b5:dc:41       no                22.22
  1     08:00:09:fb:39:a1       no                27.24
  1     08:00:09:fc:92:2c       no                53.13
  4     08:00:09:fc:d2:11       yes                0.00
  1     08:00:09:fd:23:88       no               230.42
  1     08:00:09:fe:0d:6f       no               144.55
```

❽

Sets the aging time. The aging time is the number of seconds a MAC−address will be kept in the forwarding database after having received a packet from this MAC address. The entries in the forwarding database are periodically timed out to ensure they won't stay around forever. Normally there should be no need to modify this parameter.

❾

Sets the bridge's relative priority. The bridge with the lowest priority will be elected as the root bridge. The root bridge is the "central" bridge in the spanning tree. More information about STP you find at Section 7.1.

❿

Sets the forwarding delay time. The forwarding delay time is the time spent in each of the Listening and Learning states before the Forwarding state is entered.

(11)

Sets the garbage collection interval. Every (this number) seconds, the entire forwarding database is checked for timed−out entries. The timed−out entries are removed.

(12)

Sets the hello time. Every (this number) seconds, a hello packet is sent out by the Root Bridge and the Designated Bridges. Hello packets are used to communicate information about the topology throughout the entire Bridged Local Area Network. More information about STP you find at Section 7.1.

(13)

Sets the maximum message age. If the last seen (received) hello packet is more than this number of seconds old, the bridge in question will start the takeover procedure in attempt to become the Root Bridge itself. More information about STP you find at Section 7.1.

(14)

Sets the cost of receiving (or sending, I'm not sure) a packet on this interface. Faster interfaces should have lower path costs. These values are used in the computation of the minimal spanning tree. More information about STP you find at Section 7.1. Paths with lower costs are likelier to be used in the spanning tree than high−cost paths (As an example, think of a gigabit line with a 100Mbit or 10Mbit line as a backup line. You don't want the 10/100Mbit line to become the primary line there.)

The Linux implementation currently sets the path cost of all eth* interfaces to 100, the nominal cost for a 10Mbit connection. There is unfortunately no easy way to discern 10Mbit from 100Mbit from 1Gbit Ethernet cards, so the bridge cannot use the real interface speed.

(16)

With this parameter you can enable or disable the Spanning Tree Protocol.

❾***(12)(14)(16)***

This parameters are only of interest, if you have more than one bridge in your LAN and stp enabled. Before modifying them you should read Section 7.1.

6.2. Basic Setup

The standard configuration should consist of:

1. Create the bridge interface.
   ```
   root@mbb-1:~ # brctl addbr mybridge
   ```
2. Add interfaces to the bridge.
   ```
   root@mbb-1:~ # brctl addif mybridge eth0
   root@mbb-1:~ # brctl addif mybridge eth1
   ```
3. Zero IP the interfaces.
   ```
   root@mbb-1:~ # ifconfig eth0 0.0.0.0
   root@mbb-1:~ # ifconfig eth1 0.0.0.0
   ```
4. Put up the bridge.
   ```
   root@mbb-1:~ # ifconfig mybridge up
   ```
5. Optionally you can configure the virtual interface **`mybridge`** to take part in your network. It behaves like one interface (like a normal network card). Exactly that way you configure it, replacing the previous command with something like:
   ```
   root@mbb-1:~ # ifconfig mybridge 192.168.100.5 netmask 255.255.255.0 up
   ```

A more sophisticated setup script you will find at Example 16.

> **Important:** If you get the terrible experience of a frozen system or some nasty behavior of your nicely shaped linux box at
>
> ```
> root@mbb-1:~ # ifconfig ethn 0 0.0.0.0
> ```
>
> please try (after the reboot of the system if necessary) before starting any bridge stuff at all a
>
> ```
> root@mbb-1:~ # ifconfig ethn promisc up
> ```
>
> If again your system is frozen it's you NIC's driver you have to blame, not the bridging code.

7. Advanced Bridge Features

Here we will cover some advanced features of the new bridge code.

7.1. Spanning Tree Protocol

Tell me...

- You are a networkadmin...?
- You have a switch on top of your ethernet tree...?
- You have nightmares of a switch emmiting smoke...?
- Your company is not extremely rich and con provide another redundant switch just waiting for you to plug the patchwires..?
- You don't feel like placing your bed close to your main network node to plug the wires...?

Don't wait until you're just another nervous wreck. Join linux bridge community and enjoy the relaxment a stp−enabled inhouse scenario is offering to you.

Ok, let's leave that commercial and get back linux and the bridge. Take a look on this small thread from the linux−bridge mailing list.

STP Thread from bridge@openrock.net (no more valid)

Could you give me some hints about multi−bridge scenarios.
Does the STP "heartbeat" mechanism also work with bridges with more than two cards?
How fast does it get up, and what can I do about it?

Could you give me some hints about multi−bridge scenarios.

You can just set up two "mirrored" bridges. You have two network interfaces in your bridge? Set up the mirror bridge so that it has two network interfaces as well, and connect each of the interfaces to one subnet. This will work without the need of configuration.

> **Note:** Be sure that you have the spanning tree protocol enabled. If you didn't use **brctl**, this should be fine, because in Linux, it is on by default. To check, you could check whether the bridge sends a packet to `0180c2000000` every 2 seconds. If it does, the STP is on. The STP is needed so that only one bridge will be active at any given time.

> **Note:** To be able to see nicely formatted stp packages in your network take a look at at the bridge homepage for the patches to tcpdump.

The "master" bridge will send out STP packets every 2 seconds by default. The "slave" bridge will receive these packets, and will notice that the master is still up. If the slave hasn't received a packet in 20 seconds (max. message age parameter), it will start the takeover procedure. From the moment the takeover procedure starts, it will take about 30 seconds (twice the forward delay parameter) for the bridge to become fully operational.

Does the STP "heartbeat" mechanism also work with bridges with more than two cards?

Yes, it works with any number of interfaces. You can invent bizarre topologies to your heart's desire. You can use multiple (redundant) bridge−bridge connects, you can insert loops, whatever. The STP code will always find the minimal spanning tree. The bridge code will even deal with the loss of any number of interfaces. If there are two redundant bridges with identical connections, the loss of an interface on one of the bridges will cause the other bridge to take over forwarding to that specific interface. *Now isn't that great? :)*

How fast does it get up, and what can I do about it?

If you think 50 seconds is too much −− and I guess you should; alas, the IEEE specs gives us these default values −− you can tweak these parameters. If you set the hello time (the STP packet interval) from 2 to 1 second, you can safely set the message age parameter to 4 seconds. Then you can set the forward delay to 4 seconds, and this will in total give you a takeover time of ~12 seconds.

The great thing which is made possible by STP is a redundant parallel bridging scenario, with automated take over features. Within a network basing on stp the bridges always try to send a datagram the (by path cost) shortest path. Only on that path the bridges are forwarding, all other paths between this shortest way are blocked. If there is a broken path, the bridges agree about the next shortest. So doubled paths don't break the net, but are bringing more security... For a example setup of a fail secured connection see Section 8.

7.2. Bridge And The IP−Chains

The normal idea about a bridge would not allow anything like firewalling, but since several people have asked Lennert for ipchains firewalling on bridge forwarding he implemented it.

> **Important:** If you want to do this, you will need to apply the special ip−chain−bridge−patch (also available at the bridge homepage).

As soon you have everything up correctly, the bridging code will check each to−be−forwarded packet against the ipchains chain which has the same name as the bridge.

So.. if a packet on eth0 is to be forwarded to eth1, and those interfaces are both part of the bridge group br0, the bridging code will check the packet against the chain called 'br0'.

Warning
If the chain does not exist, the packet will be forwarded. So if you want to do firewalling, you'll have to create the chain yourself.

Example 10. A Simple Bridge Firewall Setup

```
Example:
# brctl addbr br0                                  ❶
# brctl addif br0 eth0                             ❷
# brctl addif br0 eth1                             ❸
# ifconfig br0 10.0.0.254                          ❹
# ipchains −N br0                                  ❺
# ipchains −A br0 −s 10.0.0.1/8 −i eth0 −j DENY    ❻
```

❶ Creating a bridge interface named **br0**

❷❸
: Placing eth0 and eth1 into the bridge.

❹
: Assigning a regular IP address to the bridge. The IP is taken from private network 10.X.X.X (Class A).

❺
: Creating a ip−chain named **`br0`**

❶❺

Caution
It's vital to have the same name here (**`br0`** or whatever you have selected, as long as you have the same in all places).

❻
: Denying all trafic with source 10.X.X.X on eth0.

8. A Practical Setup Example

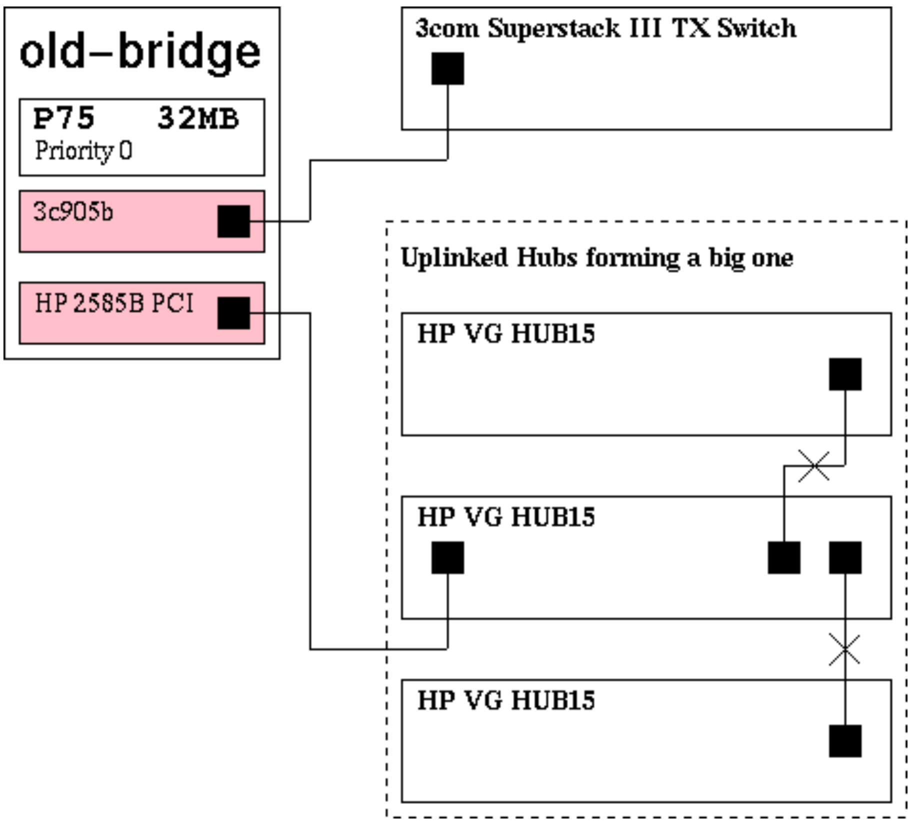
> This is a real−world example which is currently working in our network. Even if it's for sure not a very common situation it might be useful.

I had to solve a small hardware incompatibility. HP−VG (Voice Grade) 100Mbit network is not fast Ethernet compatible. Having neither the money nor the will to replace the stuff and having the need to expand the system I had to find a solution which was a) stable and b) cheap.

For sure buying a HP modular switch was not meeting condition b). So I remembered I heard about Linux−bridging which automatically fulfilled condition a) and b).

So quite some time ago I successfully set up a bridge between the two incompatible networks. Its first hardware−layout is shown in Figure 1.

Figure 1. Hardware setup Of The Old Bridge Scenario



The old setup of my previous linux bridge

It was configured as a transparent network component, meaning it didn't take a part in the network, but only bridged it. Originally it was set up on kernel 2.0.35 from a SuSE 5.3 distribution.

The next problem showed up at once. A single bridge connecting the big segments might be c) a bottleneck and d) a reason to kill the netadmin, if it blows up. So I tried to find some solution for that problem.

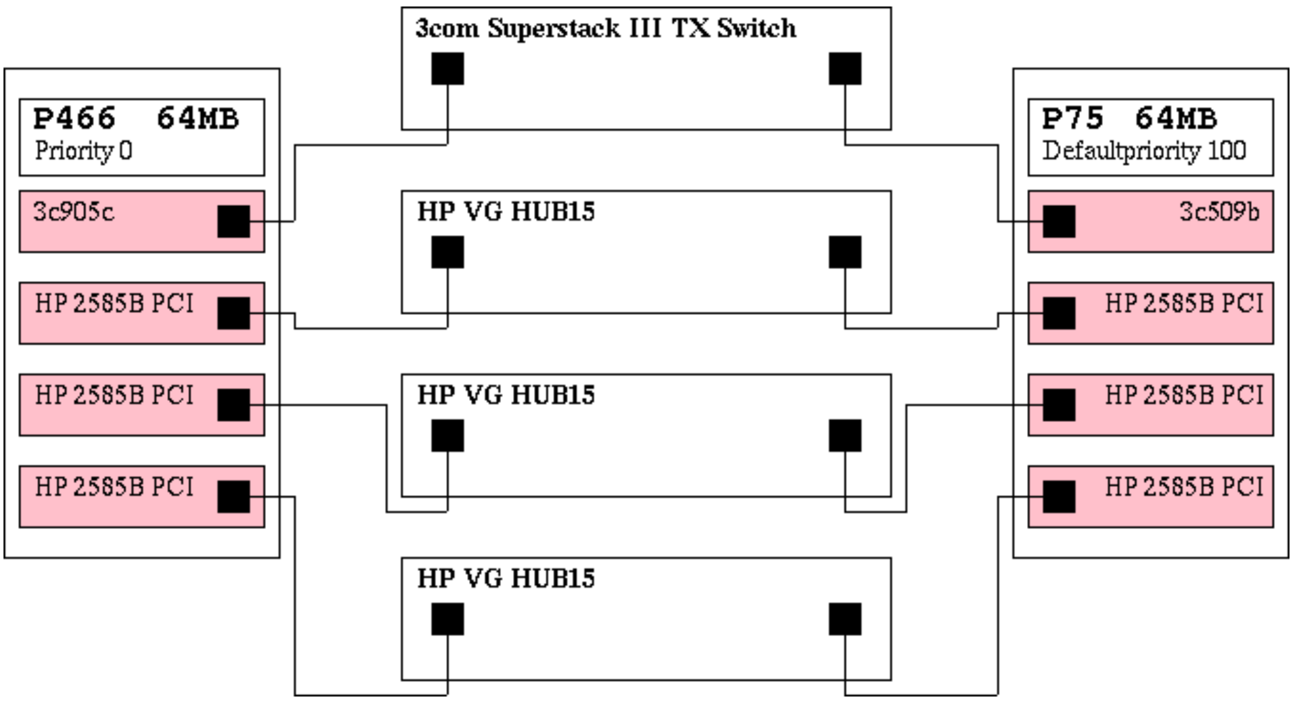
What happened next was that I discovered some hints that a new maintainer took over the bridging code. A few mails on the bridge−mailing list later as shown in Section 7.1 I was more clever. The new modular bridging code fulfilled exactly what I was looking for.

The new maintainer: Lennert Buytenhek . His project page can be found at http://www.math.leidenuniv.nl/~buytenh/bridge/ IMHO he's doing a great job. Thanks a lot.

8.1. Hardware−setup

The ideas and hints I got from the mailing list discussion shown in Section 7.1 lead to a new hardware−setup shown in Figure 2. The setup is intended to provide a default machine (guess which one). The bridge has 3 HP cards of which each is connected to a HP VG15 hub. The 3com card is connected to a 3com Superstack Fast Ethernet switch.

Figure 2. Hardware Setup Of The Multi bridge Scenario



The practically working setup of my local linux Ethernet multi bridge

This setup is not only fail proof to any one of the bridge's interfaces being down, but also to complete blackout of one of the bridges. Additional advantage to the old−setup Figure 1 that the single HUBS are switched. This means that a datagram being sent from one port on the VG15 HUB blocks 30 ports by maximum and 15 ports by minimum, instead of blocking all 45 ports. Also, the breakdown of the HUB, to the old bridge was connected, would have caused the whole HP−segment to break down. With the new code only the machines connected to the broken HUB will get no more data.

8.2. Software−setup

For both bridges the setup is exactly the same (with the exception of bridge priority which will be discussed later on). The machine was setup by the SuSE 6.4 distribution with the original unpatched kernel sources installed. At this point only the minimal configuration and no additional hardware or network setup.

The basic setup is according the descriptions in the beginning of this document. The thing I did in addition was bringing up the unpatched 2.2.14 sources of the SuSE 6.4 distribution to version 2.2.15 as in Example 11.

Example 11. Upgrading The Kernel From 2.2.14 To 2.2.15

```
root@mbb-1:~ # cd /usr/src/linux-2.2.14
root@mbb-1:/usr/src/linux-2.2.14 # patch -p1 \
    /usr/local/download/kernel/patch-2.2.15
patching file .......................
patching file ..................
...
..
root@mbb-1:/usr/src/linux-2.2.14 # cd ..
root@mbb-1:/usr/src # mv linux-2.2.14 linux-2.2.15
root@mbb-1:/usr/src # rm linux
root@mbb-1:/usr/src # ln -s linux-2.2.15 linux
```

Next step was to apply the bridge−patch as shown in Example 12.

Example 12. Applying The Kernel Patch

```
root@mbb-1:/usr/src # cd /usr/src/linux-2.2.15
root@mbb-1:/usr/src/linux-2.2.15 # patch -p1 < \
    bridge-0.0.5-against-2.2.15.diff
patching file .......................
patching file ..................
...
..
```

After that I selected the bridging code to be compiled as a module as shown in Example 13.

Example 13. Configuring The Kernel

```
root@mbb-1:/usr/src/linux-2.2.15 # make config

..

*
* Code maturity level options
*
Prompt for development and/or incomplete code/drivers (CONFIG_EXPERIMENTAL)
[N/y/?] Y
```

```
..

802.1d Ethernet Bridging (CONFIG_BRIDGE) [N/y/m/?] (NEW) m

..
```

By the way I also selected the drivers of my NIC's to be compiled as modules which resulted to `3c95x.o` and `hp100.o`.

```
root@mbb-1:/usr/src/linux-2.2.15 # make dep clean zImage \
  modules modules_install zlilo

..

root@mbb-1:/usr/src/linux-2.2.15 # init 6
```

After the reboot happening I started at runlevel 1 leaving all the networking out of the running system. That gave me the chance to check the setup step by step.

The command **modprobe −v bridge** worked without any warnings, so that one was OK. Next I edited my `/etc/modules.conf` by aliasing my network card drivers as shown in Example 14 and Example 15. I didn't need to make use of the options, all cards where realized proper as I checked by **cat /proc/modules**, **cat /proc/interrupts** and **cat /proc/ioports**.

Example 14. `/etc/modules.conf` of *mbb−1*

```
# Aliases - specify your hardware
alias eth0             3c59x
alias eth1             hp100
alias eth2             hp100
alias eth3             hp100
```

Example 15. `/etc/modules.conf` of *mbb−2*

```
# Aliases - specify your hardware
alias eth0             3c509
alias eth1             hp100
alias eth2             hp100
alias eth3             hp100
```

So next thing would have been a step by step setup of the bridge and it's interfaces. Because I'm lazy I just show the init script I prepared for the setup.

> **Important:** Of course you'll have do adapt the script to your system, if you want to use it. Please remember I'm writing this for the setup of a SuSE distribution.

Example 16. Bridge Init Script

```
#! /bin/bash
# Copyright (c) 2000 Uwe Böhme.  All rights reserved.
#
# Author: Uwe Böhme <uwe@bnhof.de>, 2000
#
#
# /sbin/init.d/bridge
#

. /etc/rc.config

return=$rc_done
case "$1" in

    start)
        echo "Starting service bridge mueb"
        brctl addbr mueb  ||  return=$rc_failed                            ❶
        brctl setbridgeprio mueb 0 || return=$rc_failed                    ❷
        brctl addif mueb eth0  ||  return=$rc_failed                       ❸
        brctl addif mueb eth1  ||  return=$rc_failed                       ❹
        brctl addif mueb eth2  ||  return=$rc_failed                       ❺
        brctl addif mueb eth3  ||  return=$rc_failed                       ❻
        ifconfig eth0 0.0.0.0  ||  return=$rc_failed                       ❼
        ifconfig eth1 0.0.0.0  ||  return=$rc_failed                       ❽
        ifconfig eth2 0.0.0.0  ||  return=$rc_failed                       ❾
        ifconfig eth3 0.0.0.0  ||  return=$rc_failed                       ❿
        brctl sethello mueb 1  ||  return=$rc_failed                       (11)
        brctl setmaxage mueb 4  ||  return=$rc_failed                      (12)
        brctl setfd mueb 4  ||  return=$rc_failed                          (13)

        echo -e "$return"
        ;;

    stop)
        echo "Shutting down service bridge mueb"
        brctl delif mueb eth3  ||  return=$rc_failed                       (14)
        brctl delif mueb eth2  ||  return=$rc_failed                       (15)
        brctl delif mueb eth1  ||  return=$rc_failed                       (16)
        brctl delif mueb eth0  ||  return=$rc_failed                       (17)
        brctl delbr mueb  ||  return=$rc_failed                            (18)
        rmmod bridge || return=$rc_failed                                  (19)

        echo -e "$return"
        ;;

    status)
        ifconfig mueb
        brctl showbr mueb
        ;;

    restart)
        $0 stop && $0 start || return=$rc_failed
        ;;

    *)
        echo "Usage: $0 {start|stop|status|restart}"
        exit 1
esac

test "$return" = "$rc_done" || exit 1
exit 0
```

❶
: This command creates a new virtual interface (bridge instance) with the name **mueb** and also brings up the bridge module.

    Note: At least my system it does. Maybe you have to enable the kernel module loader.

❷
: Here the script sets the bridge's priority (relative to other bridges in the net) to 0. This is indicating that this bridge will become the root bridge as long as there is no other bridge with a lower priority level available.

    Important: In the init script of the backup bridge this line in missing, leaving it with the default priority of 100.

❸❹❺❻
: Enslaves the Ethernet interface to become a port in the bridge.

❼❽❾❿
: Takes away any possibly disturbing IP−address and brings the interface up.

(11)
: Setting the hello time of the bridge to one second makes it possible to reduce the maxage value of the bridges inside the network.

(12)
: Setting the time the a bridge is waiting before starting the takeover process to a shorter period.

(13)
: Forcing the bridge to forward earlier than the default time.

(14)(15)(16)(17)
: Take the Ethernet out of the bridging instance.

(18)
: Destroy the bridge instance.

(19)
: Remove the bridge module.

To polish your setup and to be able to reach the bridge from remote you now can configure your bridge instance as if it would be a physical existing network interface. You can give it a nice IP with a suitable net−mask. It doesn't matter from which segment in you net, you will reach the bridge with this IP−address.

8.3. See It Work

Here I want to show and explain about how the running bridge shows up. The output Example 17 of *bridge@mbb−1* is the output of the primary bridge, while you see in Example 18 the output of the backup bridge waiting to take over.

Example 17. Status Output Of mbb−1 Fully Up

```
mueb
 bridge id              0000.0800062815f6
 designated root        0000.0800062815f6
 root port                 0                    path cost                0
 max age                   4.00                 bridge max age           4.00
 hello time                1.00                 bridge hello time        1.00
 forward delay             4.00                 bridge forward delay     4.00
```

```
 ageing time              300.00                   gc interval               4.00
 hello timer                0.80                   tcn timer                 0.00
 topology change timer      0.00                   gc timer                  3.80
 flags


eth0 (1)
 port id                  8001                     state                   forwarding
 designated root          0000.0800062815f6        path cost                100
 designated bridge        0000.0800062815f6        message age timer         0.00
 designated port          8001                     forward delay timer       0.00
 designated cost             0                     hold timer                0.80
 flags

eth1 (2)
 port id                  8002                     state                   forwarding
 designated root          0000.0800062815f6        path cost                100
 designated bridge        0000.0800062815f6        message age timer         0.00
 designated port          8002                     forward delay timer       0.00
 designated cost             0                     hold timer                0.80
 flags

eth2 (3)
 port id                  8003                     state                   forwarding
 designated root          0000.0800062815f6        path cost                100
 designated bridge        0000.0800062815f6        message age timer         0.00
 designated port          8003                     forward delay timer       0.00
 designated cost             0                     hold timer                0.80
 flags

eth3 (4)
 port id                  8004                     state                   forwarding
 designated root          0000.0800062815f6        path cost                100
 designated bridge        0000.0800062815f6        message age timer         0.00
 designated port          8004                     forward delay timer       0.00
 designated cost             0                     hold timer                0.80
 flags
```

Example 18. Status Output Of mbb−2 Fully Up

```
mueb
 bridge id                0064.00a024d04cd6
 designated root          0000.0800062815f6
 root port                   1                     path cost                100
 max age                     4.00                  bridge max age            4.00
 hello time                  1.00                  bridge hello time         1.00
 forward delay               4.00                  bridge forward delay      4.00
 ageing time               300.00                  gc interval               4.00
 hello timer                 0.00                  tcn timer                 0.00
 topology change timer       0.00                  gc timer                  2.39
 flags


eth0 (1)
 port id                  8001                     state                   forwarding
 designated root          0000.0800062815f6        path cost                100
 designated bridge        0000.0800062815f6        message age timer         0.42
 designated port          8001                     forward delay timer       0.00
 designated cost             0                     hold timer                0.00
 flags
```

```
eth1 (2)
 port id                 8002                     state                   blocking
 designated root         0000.0800062815f6        path cost                100
 designated bridge       0000.0800062815f6        message age timer          0.42
 designated port         8002                     forward delay timer        0.00
 designated cost            0                     hold timer                 0.00
 flags

eth2 (3)
 port id                 8003                     state                   blocking
 designated root         0000.0800062815f6        path cost                100
 designated bridge       0000.0800062815f6        message age timer          0.42
 designated port         8003                     forward delay timer        0.00
 designated cost            0                     hold timer                 0.00
 flags

eth3 (4)
 port id                 8004                     state                   blocking
 designated root         0000.0800062815f6        path cost                100
 designated bridge       0000.0800062815f6        message age timer          0.42
 designated port         8004                     forward delay timer        0.00
 designated cost            0                     hold timer                 0.00
 flags
```

If you take a glance into `/var/log/messages` as shown in Example 19 and in Example 20 you can see how the bridges are coming up and deciding how to do their duty. mbb−1 has a lower value for bridge−priority (see ❾), telling it to try to become the root bridge. As you can see mbb−1 forwards all ports, while mbb−2 blocks all ports with the exception of eth0.

Example 19. mbb−1 Messages From init 2

```
May 25 16:46:04 mbb-1 init: Switching to runlevel: 2
May 25 16:46:04 mbb-1 kernel: NET4: Ethernet Bridge 008 for NET4.0
May 25 16:46:04 mbb-1 kernel: device eth0 entered promiscuous mode
May 25 16:46:04 mbb-1 kernel: device eth1 entered promiscuous mode
May 25 16:46:04 mbb-1 kernel: device eth2 entered promiscuous mode
May 25 16:46:04 mbb-1 kernel: device eth3 entered promiscuous mode
May 25 16:46:04 mbb-1 kernel: mueb: port 4(eth3) entering listening state
May 25 16:46:04 mbb-1 kernel: mueb: port 3(eth2) entering listening state
May 25 16:46:04 mbb-1 kernel: mueb: port 2(eth1) entering listening state
May 25 16:46:04 mbb-1 kernel: mueb: port 1(eth0) entering listening state
May 25 16:46:08 mbb-1 kernel: mueb: port 4(eth3) entering learning state
May 25 16:46:08 mbb-1 kernel: mueb: port 3(eth2) entering learning state
May 25 16:46:08 mbb-1 kernel: mueb: port 2(eth1) entering learning state
May 25 16:46:08 mbb-1 kernel: mueb: port 1(eth0) entering learning state
May 25 16:46:12 mbb-1 kernel: mueb: port 4(eth3) entering forwarding state
May 25 16:46:12 mbb-1 kernel: mueb: topology change detected, propagating
May 25 16:46:12 mbb-1 kernel: mueb: port 3(eth2) entering forwarding state
May 25 16:46:12 mbb-1 kernel: mueb: topology change detected, propagating
May 25 16:46:12 mbb-1 kernel: mueb: port 2(eth1) entering forwarding state
May 25 16:46:12 mbb-1 kernel: mueb: topology change detected, propagating
May 25 16:46:12 mbb-1 kernel: mueb: port 1(eth0) entering forwarding state
May 25 16:46:12 mbb-1 kernel: mueb: topology change detected, propagating
```

Example 20. mbb−2 Messages From init 2

```
Jun  8 06:06:16 mbb-2 init: Switching to runlevel: 2
Jun  8 06:06:17 mbb-2 kernel: NET4: Ethernet Bridge 008 for NET4.0
Jun  8 06:06:17 mbb-2 kernel: device eth0 entered promiscuous mode
Jun  8 06:06:17 mbb-2 kernel: device eth1 entered promiscuous mode
Jun  8 06:06:17 mbb-2 kernel: device eth2 entered promiscuous mode
Jun  8 06:06:17 mbb-2 kernel: device eth3 entered promiscuous mode
Jun  8 06:06:17 mbb-2 kernel: mueb: port 4(eth3) entering listening state
Jun  8 06:06:17 mbb-2 kernel: mueb: port 3(eth2) entering listening state
Jun  8 06:06:17 mbb-2 kernel: mueb: port 2(eth1) entering listening state
Jun  8 06:06:17 mbb-2 kernel: mueb: port 1(eth0) entering listening state
Jun  8 06:06:17 mbb-2 kernel: mueb: port 2(eth1) entering blocking state
Jun  8 06:06:17 mbb-2 kernel: mueb: port 3(eth2) entering blocking state
Jun  8 06:06:17 mbb-2 kernel: mueb: port 4(eth3) entering blocking state
Jun  8 06:06:21 mbb-2 kernel: mueb: port 1(eth0) entering learning state
Jun  8 06:06:25 mbb-2 kernel: mueb: port 1(eth0) entering forwarding state
```

8.4. Bridge Tests

To check if really all the promised features are working, I did some crude test. The message logs are shown here in.

8.4.1. Tear The Patch Wire Test

I think just taking a patch wire out of a bridge port is a really good real survival test. So I pulled the plugs one by one out of the sockets and looked what happened. To give you not too much tension let me summarize first: *It's really working*. All the takeovers happened within less then 12 seconds.

The really interesting messages you can find at mbb−2. To see how everything comes up, I stopped network services first. In Example 21 you will see the messages caused by a **init 2** followed by a "take out the plug, wait what happens, then place it back" in the order eth3, eth2, eth1, eth0 .

> **Note:** The thing I did, was making the tests, and publishing the dump. The one writing the nice explanations was Lennert again.

Example 21. mbb−2 Message Output Of Bridge Test

```
Jun  8 06:06:16 mbb-2 init: Switching to runlevel: 2
Jun  8 06:06:17 mbb-2 kernel: NET4: Ethernet Bridge 008 for NET4.0
Jun  8 06:06:17 mbb-2 kernel: device eth0 entered promiscuous mode
Jun  8 06:06:17 mbb-2 kernel: device eth1 entered promiscuous mode
Jun  8 06:06:17 mbb-2 kernel: device eth2 entered promiscuous mode
Jun  8 06:06:17 mbb-2 kernel: device eth3 entered promiscuous mode
Jun  8 06:06:17 mbb-2 kernel: mueb: port 4(eth3) entering listening state
Jun  8 06:06:17 mbb-2 kernel: mueb: port 3(eth2) entering listening state
Jun  8 06:06:17 mbb-2 kernel: mueb: port 2(eth1) entering listening state
Jun  8 06:06:17 mbb-2 kernel: mueb: port 1(eth0) entering listening state
Jun  8 06:06:17 mbb-2 kernel: mueb: port 2(eth1) entering blocking state
Jun  8 06:06:17 mbb-2 kernel: mueb: port 3(eth2) entering blocking state
Jun  8 06:06:17 mbb-2 kernel: mueb: port 4(eth3) entering blocking state
Jun  8 06:06:21 mbb-2 kernel: mueb: port 1(eth0) entering learning state
Jun  8 06:06:25 mbb-2 kernel: mueb: port 1(eth0) entering forwarding state
```

```
Jun  8 06:07:15 mbb−2 kernel: mueb: neighbour 0000.08:00:06:28:15:f6 lost on port 4(eth3) ❸
Jun  8 06:07:15 mbb−2 kernel: mueb: port 4(eth3) entering listening state                 ❹
Jun  8 06:07:19 mbb−2 kernel: mueb: port 4(eth3) entering learning state                  ❺
Jun  8 06:07:23 mbb−2 kernel: mueb: port 4(eth3) entering forwarding state                ❻
Jun  8 06:07:23 mbb−2 kernel: mueb: topology change detected, sending tcn bpdu            ❼
Jun  8 06:08:51 mbb−2 kernel: mueb: topology change detected, sending tcn bpdu            ❽
Jun  8 06:08:51 mbb−2 kernel: mueb: port 4(eth3) entering blocking state                  ❾
Jun  8 06:09:22 mbb−2 kernel: mueb: neighbour 0000.08:00:06:28:15:f6 lost on port 3(eth2) ❿
Jun  8 06:09:22 mbb−2 kernel: mueb: port 3(eth2) entering listening state
Jun  8 06:09:26 mbb−2 kernel: mueb: port 3(eth2) entering learning state
Jun  8 06:09:30 mbb−2 kernel: mueb: port 3(eth2) entering forwarding state
Jun  8 06:09:30 mbb−2 kernel: mueb: topology change detected, sending tcn bpdu
Jun  8 06:10:09 mbb−2 kernel: mueb: topology change detected, sending tcn bpdu
Jun  8 06:10:09 mbb−2 kernel: mueb: port 3(eth2) entering blocking state
Jun  8 06:10:10 mbb−2 kernel: mueb: retransmitting tcn bpdu                               (11)
Jun  8 06:10:41 mbb−2 kernel: mueb: neighbour 0000.08:00:06:28:15:f6 lost on port 2(eth1) (12)
Jun  8 06:10:41 mbb−2 kernel: mueb: port 2(eth1) entering listening state
Jun  8 06:10:45 mbb−2 kernel: mueb: port 2(eth1) entering learning state
Jun  8 06:10:49 mbb−2 kernel: mueb: port 2(eth1) entering forwarding state
Jun  8 06:10:49 mbb−2 kernel: mueb: topology change detected, sending tcn bpdu
Jun  8 06:11:06 mbb−2 kernel: mueb: topology change detected, sending tcn bpdu
Jun  8 06:11:06 mbb−2 kernel: mueb: port 2(eth1) entering blocking state
Jun  8 06:11:33 mbb−2 kernel: mueb: neighbour 0000.08:00:06:28:15:f6 lost on port 1(eth0) (13)
Jun  8 06:11:33 mbb−2 kernel: mueb: port 2(eth1) entering listening state
Jun  8 06:11:37 mbb−2 kernel: mueb: port 2(eth1) entering learning state
Jun  8 06:11:41 mbb−2 kernel: mueb: port 2(eth1) entering forwarding state
Jun  8 06:11:41 mbb−2 kernel: mueb: topology change detected, sending tcn bpdu
Jun  8 06:14:18 mbb−2 kernel: mueb: topology change detected, sending tcn bpdu
Jun  8 06:14:18 mbb−2 kernel: mueb: port 2(eth1) entering blocking state
Jun  8 06:14:19 mbb−2 kernel: mueb: retransmitting tcn bpdu
```

❶

The kernel sees that there are already bridges (actually, only one of them, but Hello packets are coming in on all 4 of the ports) on eth[0123].

❷

To maintain connectivity with the rest of the network, the bridge decides to keep port 1 (eth0) active (i.e. in the "forwarding" state), and to temporarily disable ports 2−4.

❸

The plug on eth3 was pulled. Here you can see that the message age timer expired (**(13)**). The last Hello packet was seen more than X seconds ago. The bridge concludes that the connection to the bridge that was there has died. Therefore, it is going to try to enable this port, to provide network connectivity to the now−cutoff segment.

❹

It enters the listening state. It waits to see whether the old bridge might come back, or whether another bridge is going to claim takeover.

❺

Okay, no other bridge was seen. We're going to try to provide network connectivity to this segment ourselves. Which means: we're going to try and become "designated bridge" for this segment. We now enter the learning state. In this state, we only learn MAC addresses and we do not forward yet. This is because if we see an unknown destination address, we send the datagram to all ports, and this "flooding" will happen unnecessarily often if we have a empty MAC table. Therefore, we're going to fill up our MAC table with useful entries first, and this is what happens during the learning state.

❻

Okay, here we go. Pray for us.

❼

Because we took over for this segment, all communication towards this segment now goes through this bridge. This means that the topology has changed. If the topology changes, we must let all bridges now, so that they can time out stale MAC address location data quickly. This is why we send Topology Change Notification Bridge Protocol Data Units (tcn bpdus).

Apparently the root bridge immediately acknowledges this tcn bpdu in the next Hello message it sends (the protocol requires for the root bridge to acknowledge it), because this is the only such message we see.

> **Note:** In situations where you see loads of these messages, it means that the root bridge cannot acknowledge them, which probably means your root bridge has a twisted STP implementation.

(8)
: Hey, something happened again!

(9)
: Yup... eth3 came back online. The root bridge will provide connectivity for this segment again, so that we can disable this port.

(10)(12)(13)
: Same story for eth2, eth1 and eth0.

(11)
: This means the tcn bpdu wasn't acknowledged quick enough. That is why it is retransmitted.

The root bridge mbb−1 was not so chatty. It only reported some topology changes and propagated them as you can see in Example 22. If somebody can offer a explanation why the root bridge is so quiet in messaging please tell me.

Example 22. mbb−2 Message Output Of Bridge Test

```
Jun  8 06:06:52 mbb-1 kernel: mueb: received tcn bpdu on port 1(eth0)
Jun  8 06:06:52 mbb-1 kernel: mueb: topology change detected, propagating
Jun  8 06:07:31 mbb-1 kernel: mueb: received tcn bpdu on port 1(eth0)
Jun  8 06:07:31 mbb-1 kernel: mueb: topology change detected, propagating
Jun  8 06:07:32 mbb-1 kernel: mueb: received tcn bpdu on port 1(eth0)
Jun  8 06:07:32 mbb-1 kernel: mueb: topology change detected, propagating
Jun  8 06:08:11 mbb-1 kernel: mueb: received tcn bpdu on port 1(eth0)
Jun  8 06:08:11 mbb-1 kernel: mueb: topology change detected, propagating
Jun  8 06:08:29 mbb-1 kernel: mueb: received tcn bpdu on port 1(eth0)
Jun  8 06:08:29 mbb-1 kernel: mueb: topology change detected, propagating
Jun  8 06:09:03 mbb-1 kernel: mueb: received tcn bpdu on port 2(eth1)
Jun  8 06:09:03 mbb-1 kernel: mueb: topology change detected, propagating
Jun  8 06:11:40 mbb-1 kernel: mueb: received tcn bpdu on port 1(eth0)
Jun  8 06:11:40 mbb-1 kernel: mueb: topology change detected, propagating
Jun  8 06:11:41 mbb-1 kernel: mueb: received tcn bpdu on port 1(eth0)
Jun  8 06:11:41 mbb-1 kernel: mueb: topology change detected, propagating
```

One of the other bridges tells us that the topology of the LAN has changed (see Example 21). Well, okay. We will set lower timeouts on our MACC table for a short period of time, and we will propagate this topology change throughout the network.

8.4.2. Kill The Root Bridge Test

The ultimate test is of course a total blocking, breakdown or something similar to the root bridge. I did this by shooting down the root bridge by **init 1**. Next I brought it up again with **init 2**. Last I pulled all plugs out of the root bridge and waited for some time before I placed them again. In Example 23 you will see the messages from the master−bridge mbb−1, and in Example 24 you see what happened the same time at the backup−bridge mbb−2.

Example 23. Test Messages Of Master Bridge mbb−1

```
Jun 12 13:35:15 mbb-1 init: Switching to runlevel: 1
Jun 12 13:35:20 mbb-1 kernel: mueb: port 4(eth3) entering disabled state
Jun 12 13:35:20 mbb-1 kernel: mueb: port 3(eth2) entering disabled state
Jun 12 13:35:20 mbb-1 kernel: mueb: port 2(eth1) entering disabled state
Jun 12 13:35:20 mbb-1 kernel: mueb: port 1(eth0) entering disabled state
Jun 12 13:35:20 mbb-1 kernel: mueb: port 2(eth1) entering disabled state
Jun 12 13:35:20 mbb-1 kernel: device eth1 left promiscuous mode
Jun 12 13:35:20 mbb-1 kernel: mueb: port 1(eth0) entering disabled state
Jun 12 13:35:20 mbb-1 kernel: device eth0 left promiscuous mode
Jun 12 13:35:20 mbb-1 kernel: mueb: port 4(eth3) entering disabled state
Jun 12 13:35:20 mbb-1 kernel: device eth3 left promiscuous mode
Jun 12 13:35:20 mbb-1 kernel: mueb: port 3(eth2) entering disabled state
Jun 12 13:35:20 mbb-1 kernel: device eth2 left promiscuous mode
Jun 12 13:35:50 mbb-1 init: Switching to runlevel: 2
Jun 12 13:35:50 mbb-1 kernel: NET4: Ethernet Bridge 008 for NET4.0
Jun 12 13:35:51 mbb-1 kernel: device eth0 entered promiscuous mode
Jun 12 13:35:51 mbb-1 kernel: device eth1 entered promiscuous mode
Jun 12 13:35:51 mbb-1 kernel: device eth2 entered promiscuous mode
Jun 12 13:35:51 mbb-1 kernel: device eth3 entered promiscuous mode
Jun 12 13:35:51 mbb-1 kernel: mueb: port 4(eth3) entering listening state
Jun 12 13:35:51 mbb-1 kernel: mueb: port 3(eth2) entering listening state
Jun 12 13:35:51 mbb-1 kernel: mueb: port 2(eth1) entering listening state
Jun 12 13:35:51 mbb-1 kernel: mueb: port 1(eth0) entering listening state
Jun 12 13:35:51 mbb-1 kernel: mueb: received tcn bpdu on port 2(eth1)
Jun 12 13:35:51 mbb-1 kernel: mueb: topology change detected, propagating
Jun 12 13:35:52 mbb-1 kernel: mueb: received tcn bpdu on port 1(eth0)
Jun 12 13:35:52 mbb-1 kernel: mueb: topology change detected, propagating
Jun 12 13:35:55 mbb-1 kernel: mueb: port 4(eth3) entering learning state
Jun 12 13:35:55 mbb-1 kernel: mueb: port 3(eth2) entering learning state
Jun 12 13:35:55 mbb-1 kernel: mueb: port 2(eth1) entering learning state
Jun 12 13:35:55 mbb-1 kernel: mueb: port 1(eth0) entering learning state
Jun 12 13:35:59 mbb-1 kernel: mueb: port 4(eth3) entering forwarding state
Jun 12 13:35:59 mbb-1 kernel: mueb: topology change detected, propagating
Jun 12 13:35:59 mbb-1 kernel: mueb: port 3(eth2) entering forwarding state
Jun 12 13:35:59 mbb-1 kernel: mueb: topology change detected, propagating
Jun 12 13:35:59 mbb-1 kernel: mueb: port 2(eth1) entering forwarding state
Jun 12 13:35:59 mbb-1 kernel: mueb: topology change detected, propagating
Jun 12 13:35:59 mbb-1 kernel: mueb: port 1(eth0) entering forwarding state
Jun 12 13:35:59 mbb-1 kernel: mueb: topology change detected, propagating
Jun 12 13:39:03 mbb-1 kernel: mueb: received tcn bpdu on port 1(eth0)
Jun 12 13:39:03 mbb-1 kernel: mueb: topology change detected, propagating
Jun 12 13:39:05 mbb-1 kernel: mueb: received tcn bpdu on port 1(eth0)
Jun 12 13:39:05 mbb-1 kernel: mueb: topology change detected, propagating
```

Example 24. Test Messages Of Backup Bridge mbb−2

```
Jun 12 13:35:21 mbb-2 kernel: mueb: neighbour 0000.08:00:06:28:15:f6 lost on port 4(eth3)
Jun 12 13:35:21 mbb-2 kernel: mueb: port 4(eth3) entering listening state
Jun 12 13:35:21 mbb-2 kernel: mueb: neighbour 0000.08:00:06:28:15:f6 lost on port 3(eth2)
Jun 12 13:35:21 mbb-2 kernel: mueb: port 3(eth2) entering listening state
Jun 12 13:35:21 mbb-2 kernel: mueb: neighbour 0000.08:00:06:28:15:f6 lost on port 2(eth1)
Jun 12 13:35:21 mbb-2 kernel: mueb: port 2(eth1) entering listening state
Jun 12 13:35:21 mbb-2 kernel: mueb: neighbour 0000.08:00:06:28:15:f6 lost on port 1(eth0)
```

```
Jun 12 13:35:21 mbb-2 kernel: mueb: topology change detected, propagating
Jun 12 13:35:25 mbb-2 kernel: mueb: port 4(eth3) entering learning state
Jun 12 13:35:25 mbb-2 kernel: mueb: port 3(eth2) entering learning state
Jun 12 13:35:25 mbb-2 kernel: mueb: port 2(eth1) entering learning state
Jun 12 13:35:29 mbb-2 kernel: mueb: port 4(eth3) entering forwarding state
Jun 12 13:35:29 mbb-2 kernel: mueb: topology change detected, propagating
Jun 12 13:35:29 mbb-2 kernel: mueb: port 3(eth2) entering forwarding state
Jun 12 13:35:29 mbb-2 kernel: mueb: topology change detected, propagating
Jun 12 13:35:29 mbb-2 kernel: mueb: port 2(eth1) entering forwarding state
Jun 12 13:35:29 mbb-2 kernel: mueb: topology change detected, propagating
Jun 12 13:35:49 mbb-2 kernel: mueb: topology change detected, sending tcn bpdu
Jun 12 13:35:49 mbb-2 kernel: mueb: port 3(eth2) entering blocking state
Jun 12 13:35:49 mbb-2 kernel: mueb: topology change detected, \
                              <6>mueb: port 4(eth3) entering blocking state
Jun 12 13:35:49 mbb-2 kernel: mueb: topology change detected, \
                              <6>mueb: port 2(eth1) entering blocking state
Jun 12 13:35:50 mbb-2 kernel: mueb: retransmitting tcn bpdu
Jun 12 13:38:26 mbb-2 kernel: mueb: neighbour 0000.08:00:06:28:15:f6 lost on port 2(eth1)
Jun 12 13:38:26 mbb-2 kernel: mueb: port 2(eth1) entering listening state
Jun 12 13:38:27 mbb-2 kernel: mueb: neighbour 0000.08:00:06:28:15:f6 lost on port 3(eth2)
Jun 12 13:38:27 mbb-2 kernel: mueb: port 3(eth2) entering listening state
Jun 12 13:38:28 mbb-2 kernel: mueb: neighbour 0000.08:00:06:28:15:f6 lost on port 4(eth3)
Jun 12 13:38:28 mbb-2 kernel: mueb: port 4(eth3) entering listening state
Jun 12 13:38:30 mbb-2 kernel: mueb: port 2(eth1) entering learning state
Jun 12 13:38:30 mbb-2 kernel: mueb: neighbour 0000.08:00:06:28:15:f6 lost on port 1(eth0)
Jun 12 13:38:30 mbb-2 kernel: mueb: topology change detected, propagating
Jun 12 13:38:31 mbb-2 kernel: mueb: port 3(eth2) entering learning state
Jun 12 13:38:32 mbb-2 kernel: mueb: port 4(eth3) entering learning state
Jun 12 13:38:34 mbb-2 kernel: mueb: port 2(eth1) entering forwarding state
Jun 12 13:38:34 mbb-2 kernel: mueb: topology change detected, propagating
Jun 12 13:38:35 mbb-2 kernel: mueb: port 3(eth2) entering forwarding state
Jun 12 13:38:35 mbb-2 kernel: mueb: topology change detected, propagating
Jun 12 13:38:36 mbb-2 kernel: mueb: port 4(eth3) entering forwarding state
Jun 12 13:38:36 mbb-2 kernel: mueb: topology change detected, propagating
Jun 12 13:39:01 mbb-2 kernel: mueb: topology change detected, sending tcn bpdu
Jun 12 13:39:01 mbb-2 kernel: mueb: port 3(eth2) entering blocking state
Jun 12 13:39:01 mbb-2 kernel: mueb: topology change detected, \
                              <6>mueb: port 4(eth3) entering blocking state
Jun 12 13:39:02 mbb-2 kernel: mueb: topology change detected, sending tcn bpdu
Jun 12 13:39:02 mbb-2 kernel: mueb: port 2(eth1) entering blocking state
```

A. Network Interface Cards

In this section you will find a (for now) very incomplete list of NIC's which are known to work or known to cause problem. For I neither have the money to buy a lot of different NIC's, nor I have any connections to hardware vendors, I depend on your feedback to keep the list accurate. So feel free to mail about success or failure to Uwe Böhme.

Valuing Of NIC Information

− − −

Cards I tried and are also reported not to work by other people

− −

Cards I tried or are reported not to work by other people

−

Cards reported not to work by other people

+

Cards reported to work by other people

+ +

Cards I tried or are reported to work by other people

+ + +

Cards I tried and are also reported to work by other people

NIC Information

3c509b Etherlink III

+ +

3c905b/3c905c

+ + + Never heard about any problem

HP J2585A

− − System hang−up after **ifconfig**, unable to run promisc mode

HP J2585B

+ +

AMD PCnet32 10/100

+ +

RTL (Realtek) 8029

+ +

B. Recommended Reading

Here you will some recommendations which documents you should read before you start to setup a bridge.

The bridge home−page

Will give you recent information about the bridging code and the bridge utilities.

http://www.linuxdoc.org/HOWTO/NET3−4−HOWTO

Describes how to install and configure the Linux networking software and associated tools.

http://www.linuxdoc.org//HOWTO/Ethernet−HOWTO

Information about which Ethernet devices can be used for Linux, and how to set them up (focused on the hardware and low level driver aspect of the Ethernet cards).

C. FAQ

Here you will find some of the frequently asked questions connected to bridging.

FAQ

1. Hardware
- *What hardware do I need to run a bridge with 2−**`n`** NICs.*
- *Can you please recommend some tools to measure a 2−port linux bridge throughput.*

2. Software
- *I'm running with kernel x.x.x. Is a patch out there, to give me chance to use this stuff?*

1. Hardware

What hardware do I need to run a bridge with 2−**`n`** NICs.

I think a fat 486 or a modest Pentium should be able to keep up with 2x100Mbit pretty well, but I have never tested this. I don't think RAM will matter much (8 or 16MB and all should be fine). CPU will not matter a whole lot either (486/Pentium and all should be fine). I think the primary contributor is the type of bus (ISA, PCI) and the type of network cards (some network cards require less "work" than others). Big switches usually have immensely fat internal buses (3 or 4 gigabits is not uncommon). Standard PCI, for example, can't keep up with a gigabit ethernet cards.

Can you please recommend some tools to measure a 2−port linux bridge throughput.

Well, first question is: does it have 100mbit interfaces? If it hasn't (10mbit only), it shouldn't have problems with keeping up, almost regardless of the processor speed. If it does have 100mbit interfaces and you're not sure it will keep up, you can run a flood ping with big packets across it (**ping −f −s 1450 `ipaddress`**) and see whether it keeps up.

2. Software

I'm running with kernel x.x.x. Is a patch out there, to give me chance to use this stuff?

There are patches for and 2.2.14, 2.2.15. Since 2.3.47 it's in the mainstream kernel, so you don't need to patch. If you're talking about others, you will have to upgrade, if you need to bridge.

> **Note:** I've heared unconfirmed roumors about the 2.2.15 patches working without any change also with the 2.2.16 kernel. Anyone mind telling me about it?