Design Document: Using IP-Over-USB to Implement a Bridge in Linux

Ben Greenberg

February 5, 2005

Abstract

Using the `usbnet` and `g_ether` kernel modules available in the Linux 2.6 kernel, it is possible to make USB connections appear as network interfaces. Leveraging these interfaces, we can implement a fault-tolerant bridge that connects a USB network to a traditional Ethernet-based network.

Introduction

As small electronic devices become commonplace, the desire for these devices to have network connectivity is increasing. Traditionally, devices networked via IP require the use of cat5 Ethernet cable. However, the ability to fit a cat5 connector onto increasingly smaller form factors is getting difficult. Dr. Vincenzo Liberatore has suggested the use of USB to network these devices, because USB offers a "mini-B" form factor connector that is less than half the size of an Ethernet connection. However, USB was designed to connect peripherals to PCs, not carry network traffic. The Linux kernel provides a `usbnet` module that allows USB Host connections to appear as network interfaces, which has been used to interface with USB gadgets running Linux (see [2]). The kernel also provides a `g_ether` module that provides the same functionality for USB Device interfaces. As a proof of concept that Linux's IP-over-USB functionality is complete enough to be used for complex network configurations, we will construct a fault-tolerant bridge using two PCs connected via USB, connecting the systems to an existing Ethernet based network.

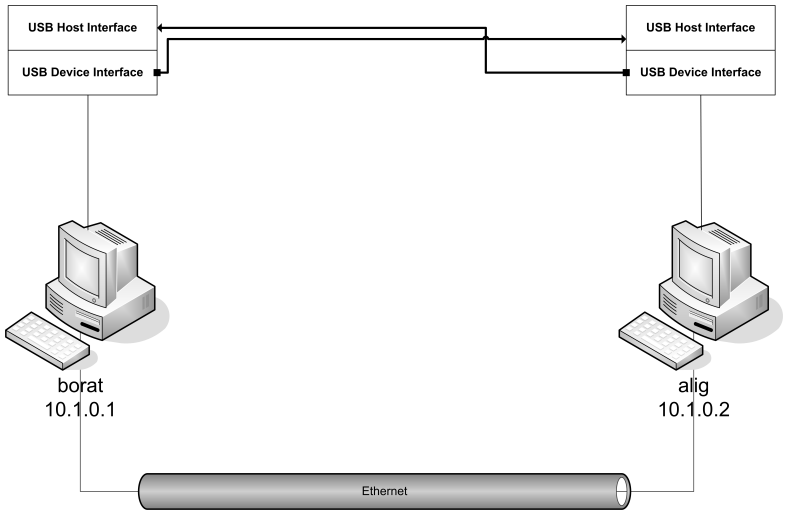

Figure 1: Physical topology

USB Background

Hardware

Our ultimate goal is to be able to network any equipment with USB support, whether it be a host or a device. For our initial project of setting up a bridge, however, we will simply network two desktop PCs. Modern desktops have USB support built into their chipsets, letting the PC act as a USB host (A-type connector). Although A to A USB cables exist, they are not within the official USB specification, and in fact the electrical connections created by such cables are not proper. To connect two desktop PCs, there are two options: a host-to-host cable with specialized electronics, or the addition of an adapter card that lets the PC act as a USB device and provides a B-type connector. The first option is well suited to ad-hoc connection of desktop PCs, as host-to-host cables are readily available from retailers and there is driver support for many operating systems. However, these cables are expensive and inflexible in that they can only be used to connect two USB hosts. These are also not included in the strict USB specification; they work by having a small USB device bridging the two hosts. Adding a USB device interface to a PC will allow us to use inexpensive, commonly available A-to-B USB cables and, since one of our long term goals is to scale down our system

into a plug-and-play embedded device, we do not want to require costly specialized hardware to connect. A USB device interface lets us program the desktop as if it was a both stand-alone USB "gadget," , and a UBS host; the two USB buses are isolated and operate independently. This allows us to develop more easily within the USB specification, which is designed for host-to-device connections. For these reasons, we installed a USB device interface card in each PC. We installed the NetChip 2280 PCI to Hi-Speed USB 2.0 Peripheral Controller [1] because of good Linux support (`net2280` driver, distributed with Linux kernel 2.4 and up) and the fact this same chipset is used in many embedded devices (combined with the availability of a Linux driver, this gives us a clear path to an embedded device).

Our setup includes two desktop machines which we will call `borat` and `alig`, both running Debian GNU/Linux 3.1[6] (aka "sarge")–this is the "testing" distribution of Debian. Linux kernel 2.6.8, not installed by default, was installed on both machines as well. The 2.6 series kernel was chosen over the more common 2.4 series for its more mature USB interfaces and drivers. We used a precompiled kernel image–Debian package `kernel-image-2.6.8-2-686`. This has all the modules we will require precompiled and ready to load as needed.

Installing the PCI USB Device interfaces was straightforward. The cards plug into an available PCI slot and provide a single USB B-type connector. The new card should be detected at boot time, which will automatically cause the `net2280` module to be loaded. If it is not detected, the module can be loaded manually by running the command `modprobe net2280` as the root user. Of course, you could also recompile your kernel with this driver built in. With this driver loaded, we can consider the hardware configured for use.

Both machines are also connected via an Ethernet network using built in NICs. The physical topology we will create is show in Figure 1.

Software

Configuring the USB device end of things (the Linux USB developers use the term "gadget" to mean a USB device) requires loading the `g_ether` module. This module allows the gadget to operate with a USB host either using the Communications Device Class (CDC) Ethernet Networking Control Model or Microsoft's RNDIS protocol. The CDC Ethernet Networking Control

Model[3] is a standardized protocol that allows USB hosts and devices to exchange Ethernet framed data. RNDIS[4] is a partially documented analogue to CDC Ethernet created by Microsoft. Since `g_ether` supports both, a gadget running this module should interface easily with Linux or Microsoft operating systems. The `g_ether` module also causes the device to properly identify itself as a CDC Ethernet device to a connected host. This allows the device to be "hotplugged" into a host, which will recognize the device and load the appropriate driver. This was confirmed in our test environment, where a host would automatically load the `usbnet` module on connection, which will enable the host-end network interface. Since both PCs have a "gadget" interface, we need to make sure that the `g_ether` module loads on both machines.

Configuring Bridging

Under Linux kernel 2.4 and up, bridging is handled by the `brctl` program with the `bridge` module[5]. To access this program under Debian, we must have the `bridge-utils` package installed. When the `brctl` program is invoked, it should automatically load the `bridge`module. The configuration shown here is done on both machines.

Setting up the bridge requires a series of commands, which could easily be put into a simple shell script. We choose to put the commands in the `/etc/network/interfaces` file instead. This allows us to bring up the bridge by running a `ifup` command (from the `ifupdown` package) and similarly bring it down using `ifdown`.

We decide to call our bridge interface `br0` on both machines and add an entry to the `/etc/network/interfaces` file describing it. The entries are identical, except for the `address` line. We are creating a 10.1.0/24 network for our machines to communicate on, we will call `borat` 10.1.0.1 and `alig` 10.1.0.2. Our configuration sets up the Ethernet connection as the root of both bridge. The `pre-up` lines specify commands to be called before the interface is brought up; the `post-down` commands are called after the interface is brought down. Note that the bridge does not need to have an IP address–it can operate invisibly. However, since we only have two machines on network, we need to be able address the machines so we can do tests.

```
iface br0 inet static
```

```
address 10.1.0.[1,2] # 1
pre-up ifconfig eth0 down # 2
pre-up brctl addbr br0 # 3
pre-up brctl addif br0 eth0 # 4
pre-up ifconfig eth0 0.0.0.0 # 5
pre-up brctl stp br0 on # 6
pre-up brctl sethello br0 1 # 7
pre-up brctl setmaxage br0 4 # 8
pre-up brctl setfd br0 4 # 9
pre-up modprobe g_ether # 10
post-down ifconfig eth0 down # 11
# should have a line to bring down all interfaces in the bridge
post-down brctl delbr br0 # 12
```

1. Configures the static IP address of each machine. We ignore the settings for gateway, broadcast, etc. because the defaults should be acceptable. `borat` will be 10.1.0.1 and `alig` will be 10.1.0.2.
2. Make sure that the Ethernet interface is not enabled; it may have been at boot.
3. Create the bridge interface.
4. Add the Ethernet interface as the root interface of the bridge.
5. Bring up the Ethernet interface without and configuration. This must be done for all interfaces on a bridge.
6. Enable the Spanning Tree Protocol on the bridge; this protocol is what provides fault tolerance to the bridge. The next 3 commands change parameters on the bridge to make fault detection and response happen more quickly. See bridge documentation[5] for more information.
7. Set HELLO time to 1 second.
8. Shorten the waiting period.
9. Force forwarding to happen earlier than default time.
10. Load the module to create a network interface for the USB device card. We will ensure that the interface is added to the bridge by modifying the network hotplug file, explained below.

11. Bring down the Ethernet interface, so that we can remove the bridge cleanly.

12. Remove the bridge from existence. We should have some extra lines to determine what interfaces are currently on the bridge and bring them all down before this called, because it will fail if any of the interfaces are still enabled.

Now that the bridge is configured, we can connect our USB cables and add the created USB interfaces. Adding interfaces to the bridge is very simple: we simply run `brctl addif br0` *`iface`* where *iface* is an existing but not configured network interface. However, we can automate this process by using the `hotplug` package. Hotplugging detects when a piece of hardware is connected, and depending on what was connected, runs a script. Hotplugging works for all kinds of devices–network, USB, PCI. We don't need to alter the USB hotplugging scripts; as discussed above, the default configuration will automatically load the `usbnet` module when the machine's host ports are connected to a CDC-enabled device. However, the `usbnet` module creates a network interface called `usb`*`i`*, and the creation of this interface will trigger the hotplug script for network devices, which is stored in `/etc/hotplug/net.agent`. The relevant portion of the file is what gets called when a new network is registered. This section begins with

```
case $ACTION in
add|register)
```

In this block, we can add code like the following:

```
case $INTERFACE in
usb*)
ifconfig $INTERFACE 0.0.0.0 && brctl addif br0 $INTERFACE
      ;;
```

which will match any interface name beginning with `usb` and run the command listed, which enables the interface without supplying any settings and, assuming it was successful, adds the interface to bridge `br0`. That's it–now, whenever a USB network interface is created, it will automatically be added to the bridge. Note that the `brctl` will fail gracefully if `br0` does not actually exist, but the interface will still be enabled in the unconfigured

state. This should probably be addressed with some more logic. Note that we loaded the `g_ether` above in our `ifupdown` script; since this creates a USB interface, it will automatically be added to the bridge to this `hotplug` configuration.

Putting it All Together

After making all these configuration changes, we can finally get the bridge up and running. The process is very simple from this point, because the configuration work we've done. Each machine should have a bridge called `br0` containing interfaces `eth0` and `usb0`. This can be confirmed with the command `brctl show`. The machines should also be able to ping each other and do any other network activity. We were able to log in to both machines via SSH through the bridges.

Now, all we have to do is connect both machines via the two USB cables. This will create a second USB network interface, `usb1`, which will be automatically added to the bridge as well. At this point, there are three possible paths for each machine the contact the other–Ethernet and two USB paths. The STP protocol figures out which paths are viable. We can also assign costs to each path; by default, all paths have the same cost, so one is not preferred over another.

Since the Ethernet interface is the root interface for the bridge (the first one added), the bridge will be using this path until is no longer viable. We can test this by simply pulling the Ethernet plug out of the computer. In our tests, the bridge noticed the failure immediately, and was able to revert to the `usb0` interface in 30-60 seconds, at which point we can contact the other machine again. We can then remove the `usb0` interface and, similarly, the bridge will begin using the other USB connection. We can also add the connections we took out; since all paths have the same cost, the bridge will only switch back to Ethernet or the other USB connection if the current connection fails.

Conclusion

The IP-over-USB functionality in the Linux kernel is mature enough to handle a fault-tolerant bridge design. With this knowledge, we hope to expand

the use of these features to provide more sophisticated bridging designs. We still need to test this type of bridge with a true USB gadget such as a PDA. There are also issues with using this type of a bridging on a DHCP enabled network that checks MAC addresses, such as CWRUnet. Since the MAC address of the sender will change when the bridge begins forwarding on a new port, some networks will get very confused and not allow the messages out. This needs to be further explored.

Bibliography

[1] http://www.plxtech.com/products/NET2000/NET2280/default.asp.

[2] David Brownell. *The GNU/Linux "usbnet" Driver.* http://www.linux-usb.org/usbnet/, March 2004.

[3] Microsoft Corporation. "universal serial bus class definitions for communications devices". http://www.usb.org/developers/devclass_docs/usbcdc11.pdf, January 1999.

[4] Microsoft Corporation. "usb remote ndis devices and windows". http://www.microsoft.com/whdc/device/network/NDIS/usbrndis.mspx, April 2004.

[5] Stephen Hemminger. *Linux Ethernet bridging.* http://bridge.sourceforge.net/index.html.

[6] Debian Project. "debian 'sarge' release information". http://www.debian.org/releases/testing/.