EE 673 Project Report

USB/IP

Rohan Mandala

Ankush Rajput

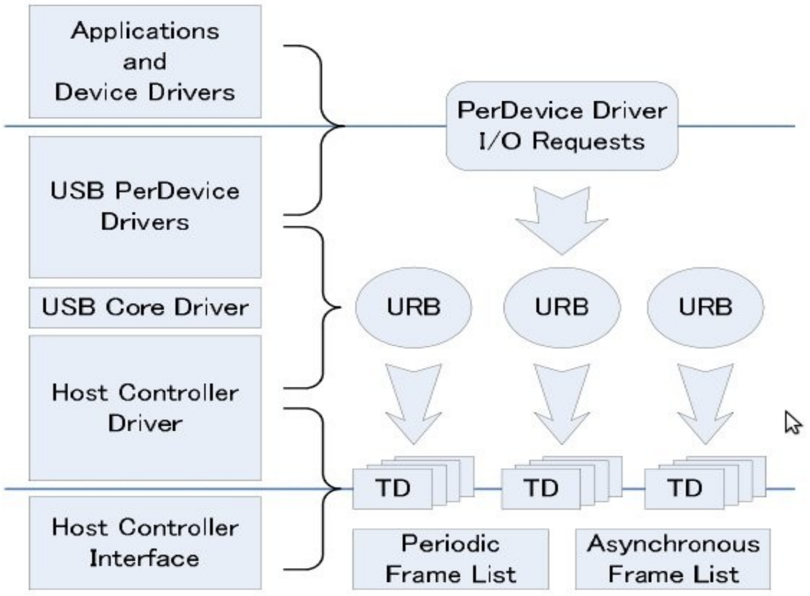
Aman Jaiswal

USB/IP – Device sharing over IP network

As personal computing becomes more popular and affordable, the availability of peripheral devices is also increasing rapidly. However, these peripheral devices can usually only be connected to a single machine at time. The ability to share peripheral devices between computers without any modification of existing computing environments is, consequently, a higly desirable goal, as it improves the efficiency of such devices.

Takira Hirocfuchi and his associates proposed USB/IP as a *peripheral bus extension* over an Internet Protocol (IP) network. This is based on the sophisticated peripheral interfaces that are supported in most modern operating systems. Using a *virtual peripheral bus driver,* users can share a diverse range of devices over networks without any modification in existing operating systems and applications. This device sharing approach presents several advantages over the conventional approaches. First, a computer can access the full functionality of a shared device. In this system, all the low level I/O data for devices are encapsulated into IP packets and then transmitted. With only a few additional drivers, users can control a shared device as if it is directly attached to a local peripheral bus.

USB MODEL



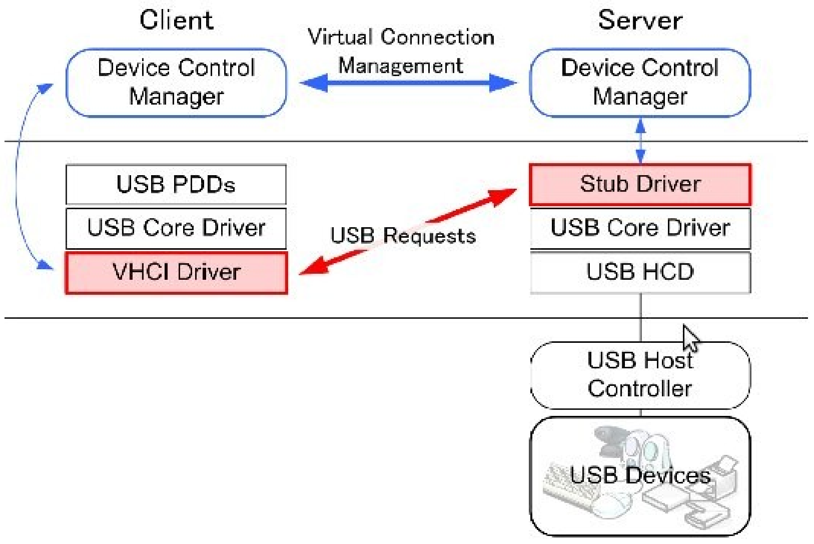
USB Device Driver Model

USB Per-Device Drivers(PDDs) are responsible for controlling individual USB devices. When applications or other device drivers request I/O to a USB device, a USB PDD converts the I/O requests to a series of USB commands and then submits them to a USB Core Driver in the form of USB Request Blocks(URBs). USB PDDs do not need to interact with the hardware interfaces of the host controllers.

USB Core Driver is responsible for the dynamic configuration and management of USB devices. When a new USB device is attached to the bus, it is enumerated by the Core Driver, which requests device specific information from it and then loads the appropriate USB PDD. A USB Core driver also provides a set of common interfaces to the upper USB PDDs and the lower USB Host Controller Drivers.

A USB Host Controller Driver (HCD) receives URBs from a USB Core Driver and then divides them into smaller requests, known as Transfer Descriptors(TDs), which correspond to the USB microframes. TD s are scheduled depending on their transfer types and are linked to the appropriate frame lists in the HCD for delivery. The actual work of the I/O transaction is perfromed by the host controller chip.

USB/IP (peripheral bus extension)



USB/IP Design

To implement the peripheral bus extension of USB, we need a Virtual Host Controller Interface(VHCI). The VHCI driver is the equivalent of a USB HCD, and is responsible for processing enqueued URBs. A URB is converted into a USB/IP request block by the VHCI driver and sent to the remote machine. A stub driver is need on the host(sever) machine which acts as a new type of USB Per-Device Driver. The stub driver is responsible for decoding incoming USB/IP packets from remote machines, extracting the URBs, and then submitting them to the local USB devices. Once a USB Core Driver enumerates and initializes the remote USB devices, unmodified PDDs and applications can access the devices as if they were locally attached.

This strategy has a problem as an IP network has a significantly larger transfer delay and more jitter than the USB network. In addition, the native I/O granularity of USB is too small to effectively control USB devices over an IP network. The isochronous transfer type(for mouse,keyboard etc.) neds to transmit 3KB of data in every microframe(125us), and the Bulk transfer type needs to transmit 6.5KB of data in a microframe. Therefore, to transfer USB commands over the IP netwrok efficiently, USB/IP is designed to encapsulate a URB(not a TD) into IP packets. This technique minimizes these timing issues by concatenating a series of USB I/O transactions into a single URB.

MULTIPLE COMPUTER ACCESS

This USB/IP implementation does not allow multiple computers to access a shared USB device simultaneously. This design criterion is logical because the USB architecture is designed to provide one computer with peripheral device access. At the beginning of USB/IP access to a shared device, a computer needs to connect the device exclusively. If the device is already used by another computer, the computer cannot use the device until another computer disconnects the device.

IMPLEMENTATION

When a USB device is plugged into a host machine, the hose automatically loads the appropriate device drivers for the USB device(hot plug and play). So inorder to implement our USB/IP , we need to first unbind(detach) the USB device from its original driver and bind it to our USB/IP stub driver. This binding/unbinding is done through the sysfs filesystem in Linux. This is done by writing the bus ID into the bind/unbind file provided by the kernel in the /sys/bus/usb/ (relaventfolder) in LINUX.

Sysfs is a feature of Linux 2.6 kernel that allows kernel code to export information to user processes via an in-memory filesystem. It is a mechanism for representing kernel objects, their attributes, and their relationships with each other. It privides two components: A kernel programming interface for exporting these items via sysfs, and a user interface to view and manipulate these items that maps back to the kernel object. We have extensively used the sysfs file system for manipulating the drivers of USB devices and their data. We have used sysfs device and driver structures in implementation. Many of the functions and structs used are in libsysfs.h and dlist.h (doubly linked list structures) from the sysfs library. Sysfs acts as a window to communicate with the kernel.

So we have to unbind the USB device from the system default driver. After binding the stub driver to the USB Device, we have to make to visible in the IP network. We write an user application for this using sockets. We deliver the URBs which will be encapsulated in IP packets through these sockets to the required client. In this project the stub driver is loaded by loading the kernel objects usbip_common.ko and usbip.ko using *insmod(* insert module). For binding/unbinding drivers we wrote an application bind.c. Stub.c receives the URBs from the stub driver and is responsible for socket connections and transmission.

On the client side, we need to load virtual host control drivers(VHCI) to handle the URBs from the remote USB device. VHCI is loaded by loading the kernel objects usbip_common.ko and vhci-hcd.ko. All the kernel modules(objects) are in the modules folder. This driver only handles the URBs. To convert the incoming IP packets into URBs and for establishing connections, we need a user application. This task is done by vhci.c

The USB/IP implementation on Linux uses a common API of a USB core driver in both client and sever sides. First, a USB PDD submits a URB by usb_submit_urb(struct *urb, ..). usb_submit_urb() calls the urb_enqueue(struct *urb, ..) of the VHCI driver after small sanity checks. The urb_enqueue() trans lates a URB into a SUBMIT PDU(Protcol Data Unit) and then transmits it to a remote Stub driver. Next , the stub driver receives the SUBMIT PDU, creates a new URB from it, and then submits the URB to a real USB host controller by usb_submit_urb(). After the requested I/O of the urb is completed, the stub driver sets up a RETURN PDU which includes the status of I/O and input data if available, and then transmits it to the VHCI driver. Finally, the VHCI DRIVER notifies the USB PDD of the completion of the URB.

All PDUs for a virtually-attached USB device are tranferred by a TCP/IP connection. In the prototype implementation, the TCP/IP connection is established using TCP sockets. To transmit the TCP/IP packets as soon as possible avoiding the buffering delay, the Nagle's algorithm is disabled. This is done using the TCP_NODELAY flag.

Continuous checking is done regarding the health of the connection. If the TCP/IP connection is suddenly disconnected, the VHCI driver detaches the device virtually and the stub driver resets the device. I/O channels (gio channels) are created on the host sockets. Polling on interrrupt devices like mouse,keyboard etc. is done from the client machine. The host machine now has no control over the USB device. The URBs from the USB device are processed on the client machine. When the client machine asks for a list of devices attached to the server, actually the device list is obtained from the usbip driver folder under the USB drivers folder in /sys file system. The I/O channels are integrated into the main event loop and any event is notified.

This implementation is working fine for interrupt devices like mouse,keyboard etc. and also for file transfers from a flash drive. For mounting the remote devices on the client system, go to the /dev file system and mount the appropriate USB device using the mount command. Opening windows of devices in nautilus may cause problems. Use the terminal for browsing through the devices.

References:

1. Takahiro Hirofuchi,Eiji Kawai, Kazutoshi Fujikawa, and Hideki Sunahara. *USB/IP – A Peripheral Bus Extension for Device Sharing over IP Network.* USENIX Annual Technical Conference, pp. 47-60, April 2005
2. Patrick Mochel. *The Sysfs filesystem.*