USB/IP - a Peripheral Bus Extension for Device Sharing over IP Network

Takahiro Hirofuchi, Eiji Kawai, Kazutoshi Fujikawa, and Hideki Sunahara
Nara Institute of Science and Technology
8916-5 Takayama, Ikoma, 630-0192, Japan
Email: {taka-hir, eiji-ka}@is.naist.jp, fujikawa@itc.naist.jp, suna@wide.ad.jp

Abstract

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 highly desirable goal, as it improves the efficiency and usability of such devices. Existing device sharing technologies in the pervasive computing area are not sufficient for peripheral devices designed for personal computers, because these technologies do not provide the degree of network-transparency necessary for both applications and device drivers.

In this paper, we propose USB/IP as a *peripheral bus extension* over an Internet Protocol (IP) network. This novel device sharing approach 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. Our experiments show that USB/IP has sufficient I/O performance for many USB devices, including isochronous ones. We also describe performance optimization criteria that can be used to achieve further performance improvements.

1 Introduction

Recent innovations in computing technology have enabled people to establish their own computing environments that comprise multiple personal computers connected together via a local area network. In such an environment, the ability to access peripheral devices attached to one computer, seamlessly and on-demand from another computer, is a highly desirable attribute. For example, a user who brings back a mobile computer to the office may want to make the backup of working files directly onto the DVD-R drive of a shared computer, rather than directly use the shared computer or move the DVD-R drive. The user may also wish to work on the mobile computer using an ergonomic keyboard and a mouse which is already attached to another a desktop computer, but without connecting to a KVM switch. In the context of resource management, the key technology for these scenarios is network-transparent device sharing by which one computer can interact seamlessly with another computer's devices in addition to directly-attached devices.

Many device sharing technologies have been proposed in the pervasive computing area, to aggregate accesses to network-attached devices and improve their usability. These technologies address dynamic discovery, on-demand selection, and automatic interaction among devices. However, little work has been done to ensure network transparency for existing device access interfaces, so that existing applications can access remote shared devices without any modification.

In this paper, we propose USB/IP as a *peripheral bus extension* over an Internet Protocol (IP) network. Our philosophy is to allow a computer to extend its local peripheral buses over an IP network to another computer. The main component of this extension is a *virtual peripheral bus driver* which provides a virtual extension of the standard peripheral bus driver. The driver resides at the lowest layer in the operating system so that most applications and device drivers can access remote shared devices through existing interfaces. We believe that this approach particularly suits recently emerging sophisticated peripheral interfaces and broadband networks.

Our device sharing approach presents several advantages over the conventional approaches. First, computers can access the full functionality of a remote shared device. The control granularity of the shared device is the same as for a directly-attached device. In our system, all the low-level control commands for a device are encapsulated into IP packets and then transmitted. Second, computers can access a shared device using standard operating systems and applications. By installing only a few additional device drivers, it is possible to control a remote device as if it was directly attached to the local peripheral bus. Third, various computers with different operating systems can share their devices with each other, because the low-level device control protocols do not depend on any operating system specific information. Last, most devices can be shared in this way, because a virtual peripheral bus driver is independent of other device drivers, and supports all the devices on its peripheral interface.

In the remainder of this paper, we expand on our vision of the peripheral bus extension. Section 2 explains the research motivation and the advantages of USB/IP from the viewpoint of the device sharing architecture. Section 3 describes the design and implementation details of USB/IP. Section 4 shows the evaluation of USB/IP and clarifies its characteristics. Section 5 provides some additional discussion and then Section 6 examines related work. We conclude our study in Section 7. The availability of USB/IP is noted in Section 8.

2 Peripheral Bus Extension

This section describes the importance of USB/IP as a new device sharing approach for peripheral bus extension. We explain what motivates the new sharing approach in contrast to conventional techniques. Finally, we explain the concept of the peripheral bus extension in the context of the device driver models of operating systems.

2.1 Motivation

The responsibility of an operating system is to manage various resources on a computer or network and to provide applications with access to these resources through generalized interfaces. A peripheral device attached to a computer is managed as one of these resources, and so is accessed by applications through a common interface which provide an abstract representation of hardware device functions (e.g., read and write). Most conventional device sharing models only allow applications to use remote devices through these abstract functions; computers do not share any more fine-grained operations than the abstract functions. Since shared devices in these models are controlled with only a few high-level operations, it is possible to manipulate the remote shared devices under narrow bandwidth restrictions, or where large network delays are present. In addition, concurrent access to a remote device can be controlled using a locking mechanism that is applied to the abstract data unit of the resource.

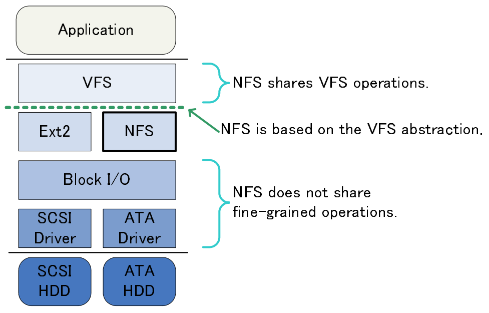

Application
VFS
NFS shares VFS operations.
Ext2
NFS
NFS is based on the VFS abstraction.
Block I/O
NFS does not share fine-grained operations.
SCSI Driver
ATA Driver
SCSI HDD
ATA HDD

Figure 1: Relation between Device Abstraction and Device Sharing (an example of storage device sharing by NFS)

Taking a storage device as an example, the most popular unit of data manipulation is a *file*. Files are organized into file systems and applications interact with them through system calls. When a hard disk is shared by NFS [15], computers control the remote shared storage device with the remote procedure calls similar to the system calls provided for direct file operations. Though the computers cannot deal with the remote storage device directly, this mechanism works fine as long as it is limited to storing and retrieving data.

While the conventional device sharing models were acceptable approaches in the past, they do not work well for sharing many of the new peripheral devices. The device sharing system should be able to maximize the use of new-featured remote devices, and also provide applications with seamless and on-demand access to both local and remote devices. However, the conventional device sharing models do not fill these requirements for the following reasons:

First, only abstract operations are sharable, so the more fine-grained and device-specific operations are not supported (Figure 1). This precludes the use of remote shared devices in the same way as directly-attached devices.

Since NFS is implemented using the virtual file system (VFS) layer in the UNIX operating system, it is not able to share either the common APIs for block devices nor the native I/O operations for ATA or SCSI disks, both of which are lower-level functions than the VFS. For example, the NFS protocol does not define methods to format a remote storage device, or to eject a remote removable media device.

Second, while the original functions of both a remote device and a locally-attached device are identical, the control interfaces for both devices are often different. Most control procedures for locally-attached devices are implemented in device drivers. However, sharing mechanisms to access remote devices are often implemented in the upper layer of the operating system, such as userland applications or libraries. This gap between both interfaces forces developers to implement a new dedicated application, or to modify existing applications to support this functionality.

For instance, VNC [12] provides framebuffer sharing with remote computers by transmitting the screen image data continuously. However, a VNC client implemented as a userland application does not provide the same access interface as the local physical framebuffer. The applications or drivers that directly handle physical framebuffers, such as console terminal drivers, cannot get the benefit of the framebuffer sharing through VNC.

Third, to achieve a high degree of interoperability is sometimes difficult for a device sharing system because of the complex differences between operating systems. Some device sharing applications (those that just extend the existing abstraction layer to forward device requests) usually only support the same operating system and cannot interoperate with other operating systems which do not have such an abstraction layer. Furthermore, the interoperability sometimes conflicts with the above first issue; the abstraction for bridging different operating systems usually disables some specific functions of shared devices.

RFS [13] provides transparent access to remote files by preserving most UNIX file system semantics, and allows all file types, including special devices and named pipes, to be shared. RFS allows access to remote devices by mounting the remote device files in the local file system tree. However, RFS is not even generic enough to connect between different versions of the UNIX operating system because of the diverse semantics of certain operations, such as `ioctl()` arguments.

Finally, peripheral devices with new features are constantly being developed. These features are often not compatible with the existing ones, and so cannot be shared without extensions to the device sharing system. In pervasive computing, the ability to adopt new technology as soon as it becomes available is a key goal. Providing remote device sharing for these new devices as soon as they are released requires a more general device sharing mechanism that can be applied to any device easily.

2.2 Peripheral Bus Extension Philosophy

In traditional operating systems, the device driver is responsible for communication with the physical device, and for providing an interface to allow userland applications to access the device functionality (Figure 2 left). This model forces conventional device sharing systems to evolve with the drawbacks described in Section 2.1.

Major advances in computer technology are now changing many of the technical issues that have compelled the conventional device sharing approach. Computer hardware is rapidly improving, and the wide availability of intelligent peripheral buses (e.g., USB and IEEE1394) is now commonplace. These peripheral buses have advanced features such as serialized I/O, plug-and-play, and universal connectivity. Serialized I/O, which supports devices with a wide range of I/O speeds, minimizes hardware and software implementation costs by reducing (or eliminating) the need for the legacy interfaces (e.g., PS/2, RS-232C, IEEE1284 and ISA). It also improves the I/O performance by employing an efficient bus arbitration mechanism for data transmission. Plug-and-play simplifies device connection by allowing dynamic attachment and automatic configuration of devices. Universal connectivity provides multiple transport modes, including isochronous transfer for multimedia devices, which improves the usability of the bus for a wide range of devices.

Device drivers are generally responsible for data transmission and communication with the attached device, while the operating system must provide the management framework for dynamic device configuration. Furthermore, a device driver is normally separated into a bus driver for manipulation of peripheral interfaces, and a per-device driver for control of devices on the peripheral interfaces. (Figure 2 center).

The USB/IP device sharing approach extends the peripheral bus over an IP network using a *virtual bus driver* (Figure 2 right). The virtual bus driver provides an interface to remote shared devices by encapsulating peripheral bus request commands in IP packets and transmitting them across the network. This approach has the advantage of being able to utilize the existing dynamic device management mechanism in the operating system, for shared devices.

Our approach resolves the drawbacks of the conventional approaches, and also has several advantages:

Full Functionality. All the functions supported by remote devices can be manipulated by the operating system. The shared operations are implemented in the lowest layer, so the abstraction at the bus driver layer conceals only the bus differences and does not affect the per-device operations. In the kernel structure, both the locally-attached devices and the remote devices are positioned in the same layer.

Network Transparency. In this paper, we define the network transparency as "shared devices on the network can be controlled by existing operating systems and applications without any modification". The virtual bus driver can conceal the implementation details of network sharing mechanisms. Shared devices are controlled by existing device drivers through the virtual bus driver. Other components of the operating system (e.g., file system, block I/O and virtual memory) and applications do not notice any difference between the access interfaces of shared devices and locally-attached ones.

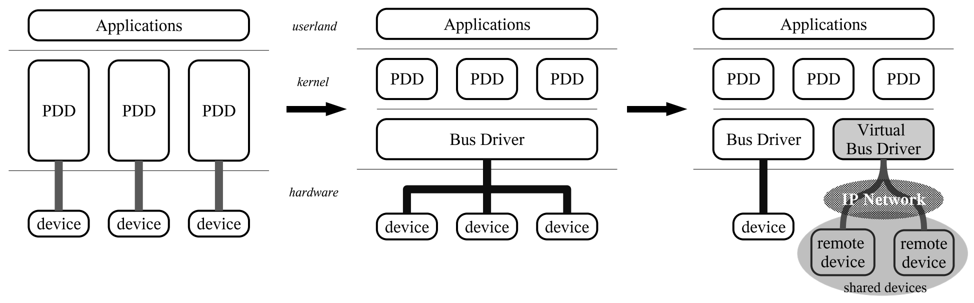


Figure 2: Evolution of Device Driver Framework
Device driver models are illustrated; conventional peripheral interfaces (left), sophisticated peripheral interfaces (center) and our proposed approach (right). PDD is the abbreviation of per-device driver.

Interoperability. The device sharing system based uses the low-level control protocols of the peripheral devices, which are the same as those of locally attached devices. These protocols, which are defined by several standards, are independent of the operating system implementation. By using these protocols for sharing, computers with different operating systems can easily share and control devices remotely.

Generality. Most devices can be shared in this way, because a virtual bus driver is independent of the per-device drivers, and supports all the devices on its peripheral interface. This allows the sharing of diverse devices over an IP network.

There are some issues that need to be considered when applying the USB/IP model in practice. First, the proposed approach is not able to provide concurrent access to a remote peripheral device, since the raw device functions are being shared. In contrast, conventional device sharing often allow concurrent access. However, our approach affects a lower layer of an operating system than the conventional approaches do; it is possible that both approaches are complementary to each other. Also, devices which support multiple simultaneous accesses, such as shared SCSI, can be shared by employing a network lock mechanism.

Second, the control protocols used by peripheral buses are not designed for transmission over IP networks. Using these protocols over an IP network raises some issues about network delay and jitters. However, high-speed network technologies, such as Gigabit Ethernet, are now able to provide bandwidths that are of the same order as those of modern peripheral interfaces. In addition, the rapid progress of networking technologies will alleviate many of these issues in the near future.

In the next section, we demonstrate the suitability of peripheral bus extension of sophisticated device interfaces by describing a prototype implementation and showing its validity through various experiments. We also discuss issues for the control of devices over IP networks that have arisen from these results.

3 USB/IP

In this section, we describe a practical example of the peripheral bus extension for the USB protocol, called USB/IP. We first discuss the USB device driver model and the granularity of operations, and then describe a strategy for IP encapsulation of the USB protocol.

3.1 USB Device Driver Model

USB (Universal Serial Bus) is one of the more sophisticated peripheral interfaces, providing serialized I/O, dynamic device configuration and universal connectivity. The USB 2.0 specification, announced in April 2000, specifies that a host computer can control various devices using 3 transfer speeds (1.5Mbps, 12.0Mbps, and 480Mbps) and 4 transfer types (Control, Bulk, Interrupt, and Isochronous).

Data transmission using the Isochronous and Interrupt transfer types is periodically scheduled. The Isochronous transfer type is used to transmit control data at a constant bit rate, which is useful for reading image data from a USB camera, or for writing sound data to a USB speaker. The shortest I/O transaction interval supported using Isochronous transfers, called a microframe, is 125us. The Interrupt transfer type negotiates the maximum delay allowed for a requested transaction. This is primarily used for USB mice and keyboards, which require a small amount of data to be sent periodically in a short interval.

The Control and Bulk transfer types are asynchronously scheduled into the bandwidth remaining after the periodic transfers have been scheduled. The Control transfer type is used primarily for enumeration and initialization of devices. In 480Mbps mode, 20% of the total bandwidth is reserved for Control transfer. The Bulk

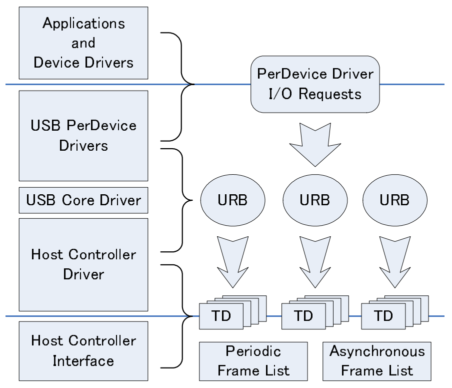


Figure 3: USB Device Driver Model

transfer type is used for the requests that have no temporal restrictions, such as storage device I/O. This is the fastest transfer mode when the bus is available.

The device driver model of USB is layered as illustrated in Figure 3. At the lower layer, the granularity of operations is fine-grained; the data size and temporal restriction of each operation are smaller than in the upper layer.

3.1.1 USB Per-Device Driver

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). A USB PDD uses only a device address, an endpoint address, an I/O buffer and some additional information required for each transfer type, to communicate with the device. USB PDDs do not need to interact with the hardware interfaces or registers of the host controllers, nor do they modify the IRQ tables.

3.1.2 USB Core Driver

A 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 PDD submits a USB request to the device via the Host Controller Driver. This request is in the form of a URB. Notification of the completed request is provided using a completion handler in the URB. This is an asynchronous process from the view point of the I/O model.

3.1.3 USB Host Controller Driver

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. TDs are scheduled depending on their transfer types and are linked to the appropriate frame lists in the HCD for delivery. TDs for Isochronous or Interrupt transfer types are linked to the Periodic Frame List and Bulk and Control transfer types are linked to the Asynchronous Frame List. The actual work of the I/O transaction is performed by the host controller chip.

3.2 IP Encapsulation Strategy

To implement the peripheral bus extension of USB, we have added a *Virtual Host Controller Interface* (VHCI) driver as a virtual bus driver (as described in Section 2.2). 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 also added as a new type of USB PDD. 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.

Using this strategy, any interface differences between directly-attached USB devices and remote USB devices is completely hidden by the HCD layer. USB PDDs, other drivers, and applications can use remote USB devices in exactly the same way. Once a USB Core Driver enumerates and initializes the remote USB devices, unmodified USB PDDs and applications can access the devices as if they were locally attached.

An IP network typically 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 needs 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 network 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. For example, using the Isochronous transfer type, by combining 80 I/O transactions that are executed every microframe into a single URB, the packet can be delayed 10ms, while still maintaining an I/O granularity of 125us. Similarly, using the Bulk transport type, a URB that has a 100KB buffer can be used to transfer 200 I/O transactions containing 512B of data each.

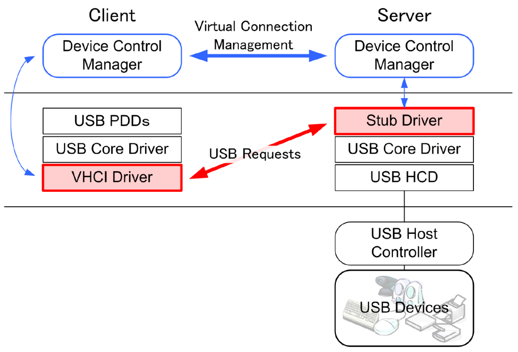


Figure 4: USB/IP Design

These issues, while important for slower networks, are minimized by new network technologies, such as Gigabit Ethernet. The bandwidth of Gigabit Ethernet is significantly greater than the 480Mbps used by USB 2.0, so it is possible to apply the URB-based I/O model to control remote USB devices over such a network. We evaluate the use of URBs for USB/IP is more detail in Section 4.

3.3 Design and Implementation of USB/IP

The design of USB/IP is illustrated in Figure 4. A VHCI driver acts as a USB HCD in the client host, and a Stub driver acts as a USB PDD in the server host. The VHCI driver emulates the USB Root Hub's behavior, so when a remote USB device is connected to a client host over the IP network, the VHCI driver notifies the USB Core Driver of the port status change. The USB/IP driver ensures that USB device numbers are translated between the client device number and the server device number. Additionally, USB requests such as `SET_ADDRESS` and `CLEAR_HALT` are intercepted so that data maintained by the USB Core Driver can be updated correctly.

Transmission of all URBs over the IP network is via the TCP protocol. However, to avoid buffering delays and transmit the TCP/IP packets as soon as possible, the Nagle algorithm is disabled. The current USB/IP implementation does not use UDP for any communication. This is because the characteristics of transmission errors for USB and UDP/IP are quite different. Though the host controller does not resubmit failed Isochronous transactions, USB PDDs and devices expect that most transactions succeed. Also, transaction failures seldom occur in USB unless there is some physical problem with devices or cables. Therefore, in general, the transport layer for URBs must guarantee in order data arrival and retransmit lost packets.

The current implementation of USB/IP supports the Linux Kernel 2.6 series. The VHCI driver and the Stub driver are implemented as loadable kernel modules. Tools used to negotiate requested devices and set up TCP/IP connections are all in userland.

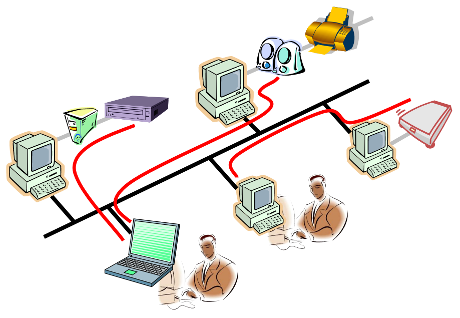


Figure 5: A USB/IP Application (Device Sharing for LAN)

```
% devconfig list              list available remote devices

  3: IO-DATA DEVICE,INC. Optical Storage
   : IP:PORT       : 10.0.0.2:3000
   : local_state   : CONN_DONE
   : remote_state  : INUSE
   : remote_user   : 10.0.0.3
   : remote_module: USBIP

  2: Logitech M4848
   : IP:PORT       : 10.0.0.2:3000
   : local_state   : DISCONN_DONE
   : remote_state  : AVAIL
   : remote_user   : NOBODY
   : remote_module: USBIP

% devconfig up 3                   attach a remote DVD Drive
% ...
% ...                        mount/read/umount a DVD-ROM
% ...                                      play a DVD movie
% ...                           record data to a DVD-R media
% ...
% devconfig down 3                 detach a remote DVD Drive
```

Figure 6: USB/IP Application Usage

As a practical application of USB/IP, we have also developed a device sharing system for LAN environments (Figure 5). A key design of the device sharing system is the support of multiple virtual buses in the operating system. A virtual bus is implemented for each sophisticated peripheral bus. For service discovery, Multicast DNS [3] and DNS Service Discovery [2] are used to manage the dynamic device name space in a local area network. An example usage of this application is illustrated in Figure 6. This paper will focus primarily on the implementation of USB/IP. Details of the application of USB/IP for device sharing will be given in another paper.

4 Evaluation

In this section, we describe the characteristics of the USB/IP implementation. In particular, we show the results of several experiments that were carried out to measure the USB/IP performance. The computers used for the evaluation are listed in Table 1. To emulate various network conditions, we used the NIST Net [1] package on Linux Kernel 2.4.18. A client machine and a server machine were connected via a NIST Net machine, as shown in Figure 7. Both the client and server machines

Table 1: Machine Specifications for Experiments

CPU	Intel Pentium III 1GHz
Memory	SDRAM 512MB
NICs (client/server)	NetGear GA302T
NICs (NIST Net)	NetGear GA620
USB 2.0 Interface	NEC μPD720100

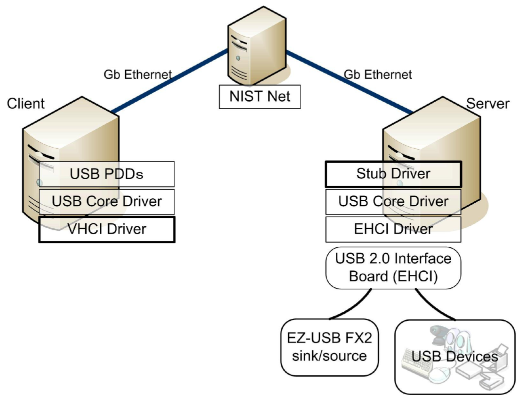


Figure 7: Experiment Environment

run Linux Kernel 2.6.8 with the USB/IP kernel modules installed.

4.1 Performance Evaluation of USB Pure Sink/Source

In the first evaluation, we used a pure sink/source USB device rather than a real USB device in order to identify the performance characteristics of USB/IP itself. A USB peripheral development board with a Cypress Semiconductor EZ-USB FX2 chip [5] was programmed to be a sink/source for the USB data. The firmware and test device driver were implemented as a USB PDD for the experiments.

4.1.1 Bulk Transfer

The I/O performance of USB/IP depends to a large degree on network delay and the data size of the operation being performed. In the second evaluation, we derived a throughput model for USB/IP from the results of the experiments, and then determined a set of criteria for optimizing USB/IP performance.

The first step of the evaluation was to measure the USB/IP overhead for USB requests of the Bulk transfer type. As Figure 8 shows, the test driver submits a Bulk URB to the remote source USB device, waits for the request to be completed, and then resubmits the request continuously. As shown in the figure, the enqueued URBs are transferred to the server's HCD between 1 and 2, and vice versa between 3 and 4. The execution time

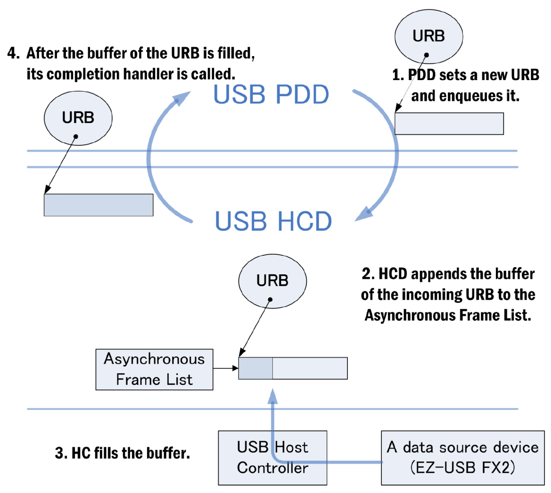

URB
4. After the buffer of the URB is filled, its completion handler is called.
1. PDD sets a new URB and enqueues it.
USB PDD
URB
USB HCD
URB
2. HCD appends the buffer of the incoming URB to the Asynchronous Frame List.
Asynchronous Frame List
3. HC fills the buffer.
USB Host Controller
A data source device (EZ-USB FX2)

Figure 8: Summarized Driver Behavior of the Experiments in Section 4.1.1

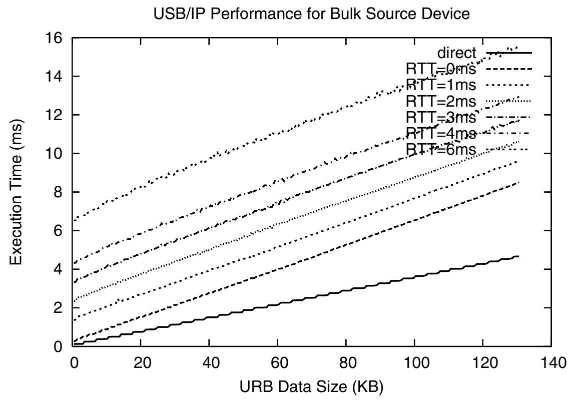


Figure 9: Execution Time of a URB

for processing a URB in the client for various values of URB data size and network round trip time (RTT) was measured using the Time Stamp Counter (TSC) register available on Intel Pentium processors. Note that when NIST Net sets the network RTT to 0ms, the actual RTT between the client and server machines is 0.12ms, as determined by `ping`.

The results are shown in Figure 9. From the graph, it can be seen that the relationship between the execution time of URBs and different data sizes is linear with constant gradient. The CPU cost for the USB/IP encapsulation is quite low at only a few percent. TCP/IP buffering does not influence the results because we use the `TCP_NODELAY` socket option. From the graph, the execution time t_{overIP} for data size s is given by

$$t_{overIP} = a_{overIP} \times s + RTT.$$

where a_{overIP} is the gradient value for the different

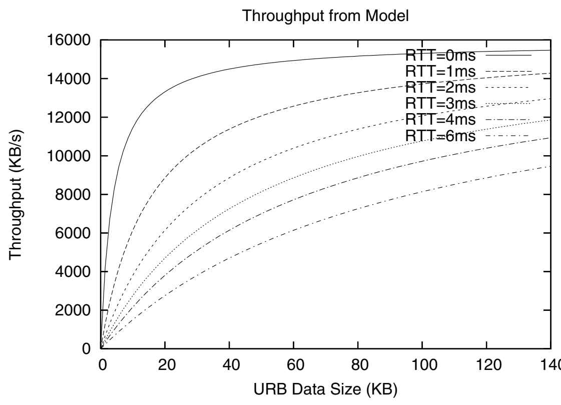


Figure 10: Throughput from Model

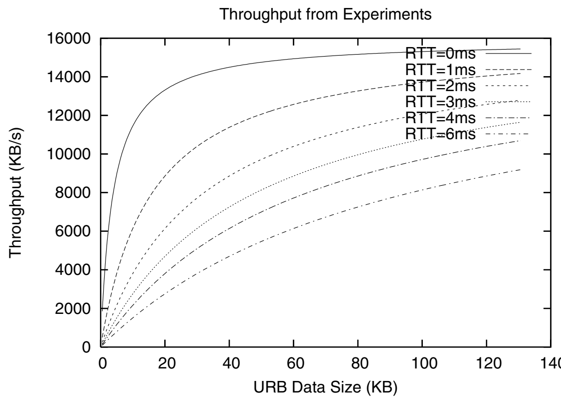


Figure 11: Throughput from Experiments

RTTs. The throughput *thpt* is then

$$thpt = \frac{s}{t_{overIP}} = \frac{s}{a_{overIP} \times s + RTT}. \quad (1)$$

A regression analysis shows a_{overIP} is 6.30e-2 ms/KB with a y-intercept for the 0ms case of 0.24 ms. The throughput modeled by Equation (1) is shown in Figure 10. The actual throughput from the experiments is illustrated in Figure 11, showing that the throughput of the model is fully substantiated by the experimental results. Therefore, within the parameter range of the experiments, this model is an accurate estimate of throughput for different URB data sizes and network RTTs.

In the directly-attached case, a_{direct} is 3.51e-2 ms/KB with a y-intercept of 0.07ms. This implies a relatively constant throughput of approximately 28.5MB/s except for quite small URB data sizes. This value is also determined largely by the performance of the host controller. The host controller in these experiments can process 6 or 7 Bulk I/O transactions per microframe (125us). In 480Mbps mode, one Bulk I/O transaction transfers 512B of data to a USB device. In this case, the throughput is equal to $(7 \times 512B)/125us = 28MB/s$.

To summarize these experiments, we have estimated

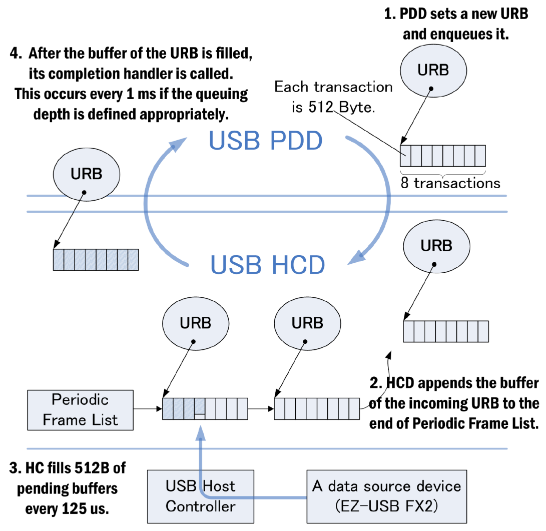

1. PDD sets a new URB and enqueues it.
4. After the buffer of the URB is filled, its completion handler is called. This occurs every 1 ms if the queuing depth is defined appropriately.
Each transaction is 512 Byte.
URB
USB PDD
8 transactions
URB
USB HCD
URB
URB
URB
Periodic Frame List
2. HCD appends the buffer of the incoming URB to the end of Periodic Frame List.
3. HC fills 512B of pending buffers every 125 us.
USB Host Controller
A data source device (EZ-USB FX2)

Figure 12: Summarized Driver Behavior of the Experiments in Section 4.1.2

the appropriate URB data size under various network delays. We have also confirmed that when multiple URBs were queued simultaneously, the throughput of Bulk transfer is dependent on the total data size of simultaneously-queued URBs. To maintain throughput when there is some network delay, USB PDDs should either enlarge each URB data size, or increase the queuing depth of URBs. Moreover, in situations when a large number of URBs are queued asynchronously, and there is substantial network delay, the TCP/IP window size must also be increased to ensure that the network pipe remains full.

4.1.2 Isochronous Transfer

In this section, we examine the performance of USB/IP when employing the isochronous transfer type. To ensure transfers for USB devices meet the Isochronous requirements, starvation of transaction requests must be avoided in the host controller. To achieve this, the USB driver model allows USB PDDs to queue multiple URBs simultaneously. In the case of USB/IP, it is also important to select a URB queuing depth that matches the likely delays introduced by the IP network.

For this test, we developed the firmware and a test device driver for an Isochronous source device. The device and drivers are configured as follows:

- A transaction moves 512B data in one microframe (125us).
- A URB represents 8 transactions.

In this case, the completion handler of the URB is called every 1ms ($125us \times 8$). This 1ms interval was chosen to be small enough so that it would be possible to examine the isochrony of USB/IP. In general, the interval of completion is set to approximately 10ms, which is an acceptable trade-off between smoothness of I/O operations and the processing cost. Figure 12 shows the detail of the driver behavior in the experiments. The USB PDD sets up each URB with the pointer to an I/O buffer for 8 transactions, and then queues the multiple URBs. The host controller then keeps pending I/O buffers on the Periodic Frame List, and the completion handler is called every 1ms. The HCD moves the input data to the USB PDD periodically. If there are no pending I/O buffers in the Periodic Frame List, the host controller does not copy data and isochronous data will be lost.

For a directly-attached source device, and the USB PDD submitting only one URB, the completion interval was 11.1ms because of request starvation. When the USB PDD submitted 2 or more URBs simultaneously, the completion intervals were 1ms, with a standard deviation of approximately 20ns for any queuing depth.

Figure 13 illustrates the mean completion intervals for various network RTTs and the queuing depths of submitted URBs for USB/IP. This shows that even under some network delays, the USB PDD is able to achieve 1ms completion intervals, provided that an adequate queuing depth is specified. For example, when the network delay is 8ms, the appropriate queuing depth is 10 or more. Figure 14 shows the time series of completion intervals in this case. Immediately after the start of the transfer, the completion intervals vary widely because the host controller does not have enough URBs to avoid starvation. Once enough URBs are available, the cycle becomes stable and the completion intervals remain at 1ms. Figure 15 shows the standard deviations of the completion intervals. With the sufficient URBs, the measured standard deviation is less than 10us, including the NIST Net's deviations [1]. These values are less than one microframe interval (125us) and adequate for most device drivers. This is the case for the Linux kernel 2.6 series, where process scheduling is driven by `jiffies`, which are incremented every 1ms. TCP/IP buffering has no impact on the timing, since the socket option `TCP_NODELAY` is set.

USB/IP periodic transfers are illustrated in Figure 16. In this case, the USB PDD in the client host queues 3 URBs which are completed every x ms. For the corresponding periodic completions in the server, the host controller must always keep multiple URBs queued. Therefore, with a queuing depth q, the time in which the next URB is pending is

$$t_{npending} = (q-1)x - RTT > 0.$$

The appropriate queuing depth q can then be calculated

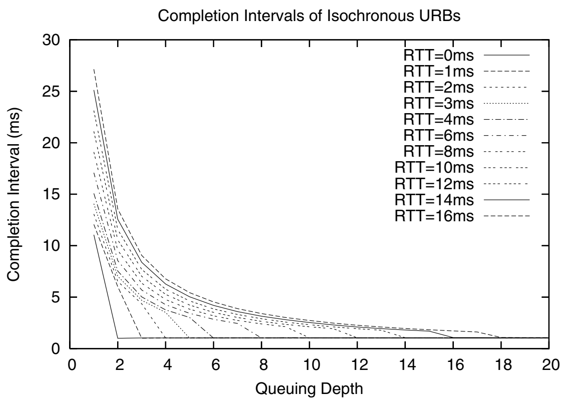


Figure 13: Mean Completion Intervals

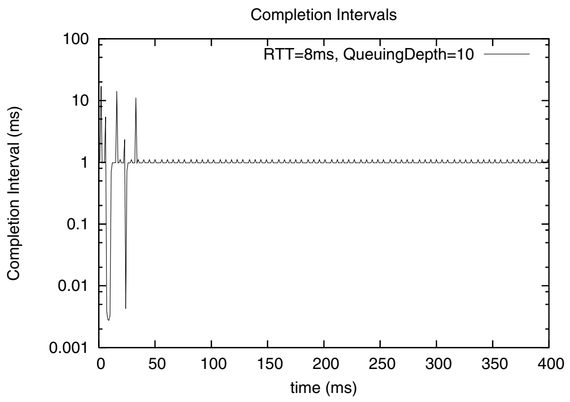


Figure 14: Time Series Data of Completion Intervals (RTT=8ms, Queuing Depth=10)

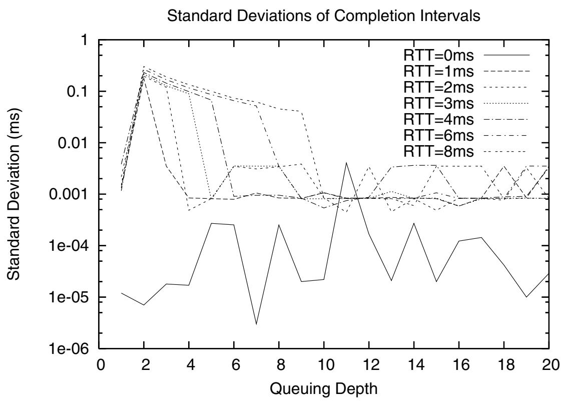


Figure 15: Standard Deviations of Completion Intervals

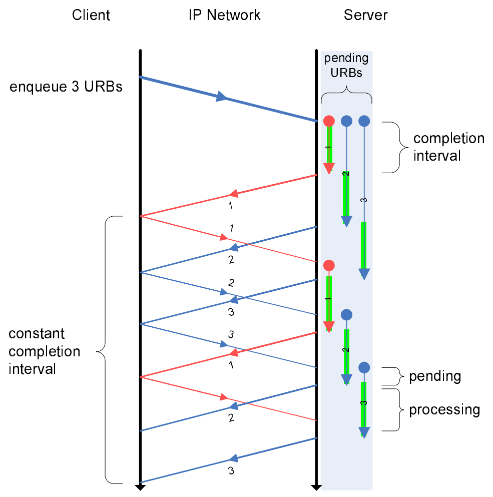


Figure 16: USB/IP Model for Periodical Transfers

by

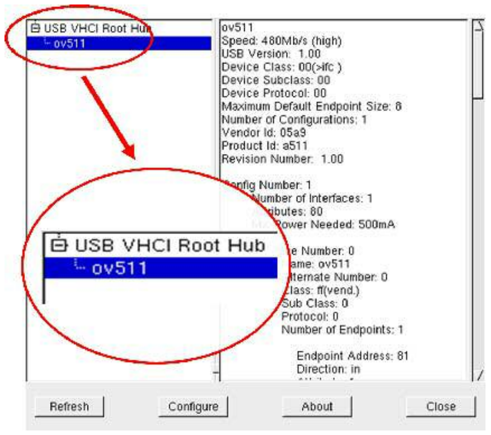
$$q > \frac{RTT}{x} + 1. \qquad (2)$$

Comparing Figure 13 and Equation (2) shows the required queuing depth of URBs in the experiments.

To summarize these experiments, USB PDDs with periodic transfers must queue multiple URBs to at least the depth q of Equation (2) to ensure a continuous stream of I/O. In the wide area networks where there is significant jitter or packet loss, q should be increased to ensure a sufficient margin is available. The result can be also applied to Interrupt transfer type URBs, which specify the maximum delay of completion. This examination continues for common USB devices over an IP network in Section 4.2.2.

4.2 Performance Evaluation of USB Devices

In this section, we examine the USB/IP characteristics for common USB devices. All USB devices we have tested can be used as USB/IP devices. Figure 17 shows a client host attached to a remote USB camera through the VHCI driver. In this case, the USB device viewer `usbview` [10] sees the device descriptors as if the camera were locally attached. The only difference that is apparent between USB and USB/IP is that the host controller is VHCI. USB PDDs can also control their corresponding remote USB devices without any modification. In our LAN environment, the performance degradation of USB/IP is negligible. Specific details of the performance of each kind of USB/IP device are described in more detail below.



```
ov511
Speed: 480Mb/s (high)
USB Version:  1.00
Device Class: 00(>ifc )
Device Subclass: 00
Device Protocol: 00
Maximum Default Endpoint Size: 8
Number of Configurations: 1
Vendor Id: 05a9
Product Id: a511
Revision Number:  1.00

onfig Number: 1
    Number of Interfaces: 1
      ibutes: 80
      ower Needed: 500mA

      e Number: 0
      ame: ov511
      lternate Number: 0
      lass: ff(vend.)
      Sub Class: 0
      Protocol: 0
      Number of Endpoints: 1

         Endpoint Address: 81
         Direction: in
```

Figure 17: `usbview` Output for a USB/IP Device

Table 2: Specifications of the Tested USB Hard Disk

Product Name	IO-DATA HDA-iU120
Interface	USB 2.0/1.1, iConnect
Capacity	120GB
Rotation Speed	5400rpm
Cache Size	2MB

4.2.1 USB Bulk Device

USB storage devices (e.g., hard disks, DVD-ROM drives, and memory drives), USB printers, USB scanners and USB Ethernet devices all use the USB Bulk transfer type. All these devices are supported by USB/IP. For USB storage devices, it is possible to create partitions and file systems, perform mount/umount operations, and perform normal file operations. Moreover, we have shown that it is possible to play DVD videos and to write DVD-R media in a remote DVD drive, using existing, unmodified, applications. As described in Section 4.1.1, the USB/IP performance of USB Bulk devices depends on the queuing strategy of Bulk URBs. We have tested the original USB storage driver of Linux Kernel 2.6.8 to show its effectiveness for USB/IP.

The experimental setup for this test was the same as that described in Section 4.1.1. NIST Net was used to emulate various network delays. We used the Bonnie++ 1.03 [4] benchmarks for ext3 file system on a USB hard disk. The disk specifications are shown in Table 2. The Bonnie++ benchmarks measure the performance of hard drives and file systems using file I/O and creation/deletion tests. The file I/O tests measure sequential I/O per character and per block, and random seeks. The file creation/deletion tests execute `creat/stat/unlink` file system operations on a large number of small files.

Figure 18 and Figure 19 show the sequential I/O throughput and the corresponding CPU usage for USB and USB/IP respectively. Figure 20 shows sequential

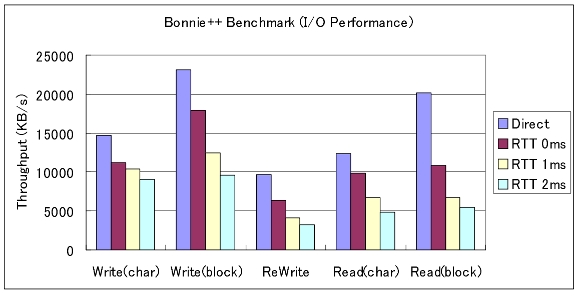


Figure 18: Bonnie++ Benchmark (Sequential Read/Write Throughput on USB and USB/IP)

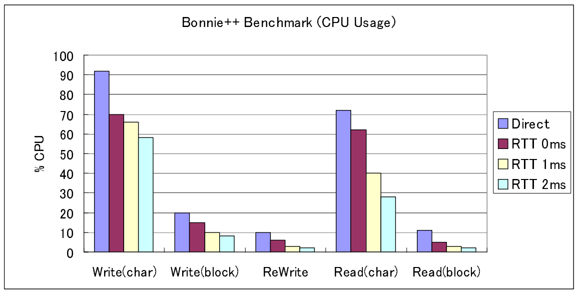


Figure 19: Bonnie++ Benchmark (Sequential Read/Write CPU Usage on USB and USB/IP)

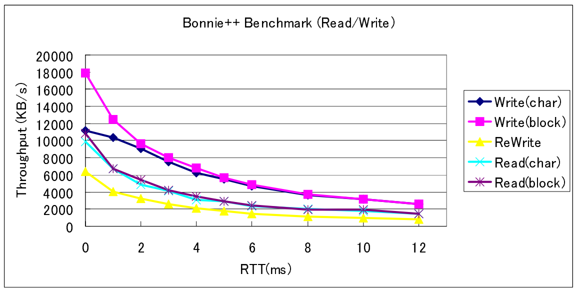


Figure 20: Bonnie++ Benchmark (Sequential Read/Write Throughput on USB/IP)

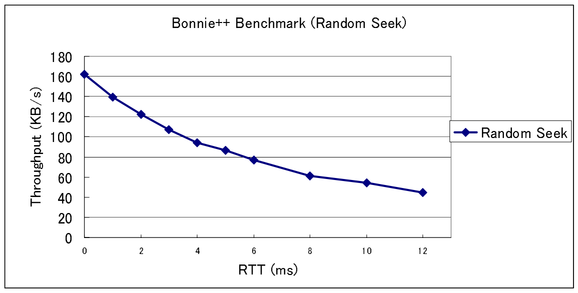


Figure 21: Bonnie++ Benchmark (Random Seek Speed on USB/IP)

I/O throughput by USB/IP under various network delays. For character write and block write, the throughput obtained by USB/IP when NIST Net's RTT is 0ms (0.12ms by `ping`) is approximately 77% of the throughput obtained by USB. Rewrite speeds are 66%, character read 79%, and block read 54% of that obtained by USB respectively. Since the CPU usage for USB/IP is less than those for USB, the bottleneck primarily results from insufficient queuing data size for the I/Os to the remote hard disk.

The Linux USB storage driver is implemented as a glue driver between the USB and SCSI driver stacks. For the SCSI stack, the USB storage driver is a SCSI host driver. A SCSI request with scatter-gather lists is repacked into several URBs, which are responsible for each scatter-gather buffer. The Linux USB storage driver does not support the queuing of multiple SCSI requests. Therefore, the total I/O data size of URBs submitted simultaneously is the same as each SCSI request size. In the case of block write, this is approximately 128KB. This queuing data size is small for USB/IP under some network delays, as we discussed in Section 4.1.1. To optimize the sequential I/O throughput for USB/IP, a reasonable solution is for the USB storage driver to provide SCSI request queuing.

The throughput of random seek I/O by USB/IP is illustrated in Figure 21. This test runs a total of 8000 random `lseek` operations on a file, using three separate processes. Each process repeatedly reads a block, and then writes the block back 10% of the time. The throughput obtained by USB for this test is 167KB/s. The throughput difference between USB and USB/IP is much smaller than that of sequential I/Os. The CPU usage in both the USB and USB/IP cases is 0%. This is because the bottleneck for random seek I/O is the seek speed of the USB hard disk itself, and is slower than that of read/write I/Os. The rational speed of the USB hard disk we tested is 5400rpm and its seek speed is approximately 10ms.

Figure 22 shows the speed of file creation and deletion operations by USB/IP under various network delays.

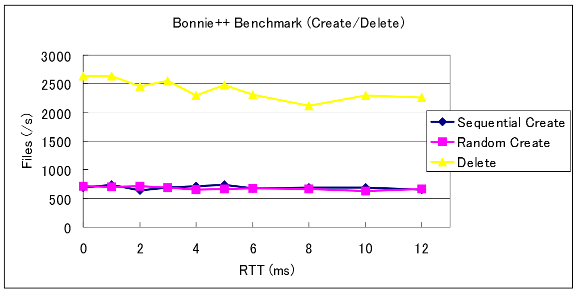


Figure 22: Bonnie++ Benchmark (Create/Delete Speed on USB/IP)

The speeds obtained by USB are 682 op/s for sequential creation, 719 op/s for random creation, and 2687 op/s for file deletion. In all these cases, the CPU usage was over 90%. The difference between USB and USB/IP is quite small, and the results of each test are almost constant under various network delays. The bottleneck for the file creation/deletion tests is predominantly the CPU resources.

4.2.2 USB Isochronous Device

USB multimedia devices, such as USB cameras and USB speakers, use the USB Isochronous transfer type to transmit data at periodic intervals. We have tested a USB camera (which uses the OmniVison OV511 chip), and a USB Audio Class speaker. These devices work completely transparently using USB/IP on a LAN. We were able to demonstrate video capture from the camera, and successfully played music using the speaker system.

The Linux USB Audio Class driver employs multibuffering by submitting 2 URBs simultaneously, where each URB is responsible for 5ms of transactions. Equation (2) shows this driver will work with a remote USB audio device provided that the network delay is 5ms or less. For larger network delays, it is still possible to use a remote audio device by increasing the completion interval for each URB. The main drawback with this is that it can result in degraded I/O response.

4.2.3 USB Interrupt Device

USB Human Input Devices, such as USB keyboards and USB mice, use the USB Interrupt transfer type to transmit data at periodic intervals, in a similar manner to interrupt requests (IRQs). Other devices also use the Interrupt transfer type to notify hosts of status changes. On our test LAN, we were able to demonstrate the correct operation of such devices using USB/IP for both consoles and the X Window System.

Most USB HID drivers submit only one URB with a completion delay of 10ms. After the URB processing is completed, the driver resubmits the URB. The drivers read the interrupt data, which is accumulated in the device endpoint buffer, every 10ms. Under large network delays, it is possible that the device endpoint buffer may overflow. When network delays approach 100ms, there is likely to be significant degradation in the performance of these devices. The former problem can be resolved by queuing more URBs so that endpoint buffer overflow is prevented. The latter problem is an underlying issue that results from attempting human interaction over a network.

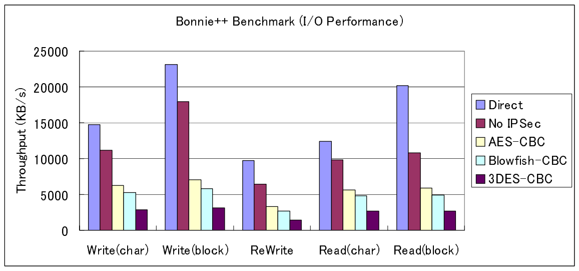


Figure 23: Bonnie++ Benchmark (Sequential Read/Write Throughput on USB and USB/IP with IPSec)

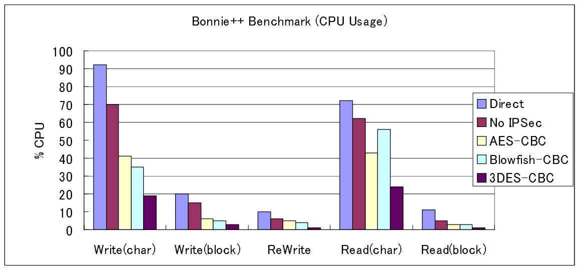


Figure 24: Bonnie++ Benchmark (Sequential Read/Write CPU Usage on USB and USB/IP with IPSec)

5 Discussion

Security. Support for authentication and security schemes is an important part of the USB/IP architecture. As most of USB/IP is currently implemented in the kernel (to avoid memory copy overhead), it is logical that an existing kernel-based security mechanism be employed. IPSec [9], which provides a range of security services at the IP layer, is one of the most suitable technologies for this purpose. IPSec provides the following functionality: access control, connectionless integrity, data origin authentication, protection against replay attacks (a form of partial sequence integrity), confidentiality (encryption), and limited traffic flow confidentiality. Figure 23 and Figure 24 show the results of the I/O benchmark described in Section 4.2.1, but with IPSec ESP (Encapsulating Security Payload) employed for all traffic between the client and the server. The hash algorithm used is HMAC-SHA1 and the encryption algorithms are AES-CBC, Blowfish-CBC, and 3DES-CBC, respectively. Since encryption entails a significant CPU overhead, the performance degradation compared to non-encrypted transfer is quite high. In the case of AES-CBC encryption, the performance is less than 50% of that obtained without IPSec. To achieve significantly higher throughput, such as would be necessary for embedded computers, some form of hardware acceleration is required. The optimization criteria, described in Section 4.1.1, may also be applied to determine the likely impact of IPSec overhead on the operation of USB/IP. To do this, IPSec overhead is considered as an estimate of pseudo network delay. The issue of authentication and security for USB/IP is an ongoing area of research, and will be dealt with more fully in a future paper.

Error Recovery. The issue of error recovery is an area that also needs to be considered for USB/IP. The error recovery methodology employed by USB/IP exploits the semantics of error recovery in the USB protocol. A dropped TCP/IP connection to a remote USB device is detected by both the VHCI driver and the Stub driver. The VHCI driver detaches the device, so it appears that the device has been disconnected, and the Stub driver resets the device. As with directly-attached USB devices that are disconnected, some applications and drivers may lose data. This recovery policy is appropriate in LAN environments, because sudden disconnection of TCP/IP sessions seldom occur. In addition, this error recovery policy greatly simplifies the USB/IP implementation. To use USB/IP with more unstable network environments, such as mobile networks, a more dependable recovery scheme is required. This is also an area of future research.

Interoperability. The USB/IP architecture has been designed so that it is interoperable between different operating systems. USB driver stacks for many operating systems are very similar. In most cases the USB driver stacks are designed with three layers: the USB PDDs, the USB HCDs, and a glue layer. In addition, the Linux, Microsoft Windows, and FreeBSD operating systems also have similar USB request structures, (`urb`, `URB` and `usbd_xfer` respectively), which means that USB/IP will be relatively easy to port. One area that this paper does not focus on is details of the interoperability features of the USB/IP protocol itself. Future work will be required to develop the interoperable implementations for these other operating systems.

6 Related Work

iSCSI [14] is designed to transport SCSI packets over a TCP/IP network, and provide access to remote storage devices. This protocol is commonly regarded as a fundamental technology for the support of SANs (Storage Area Networks). However, because iSCSI is the extension of a SCSI bus over an IP network, the protocol has been designed to ensure network transparency and interoperability. This means that the protocol could be applied to storage device sharing between computers using our peripheral bus extension technique. Our device sharing methodology is designed to provide virtual connections that are independent of device control details, so it would be possible to utilize the iSCSI protocol, as one of the control modules for the remote device. The main limitation of iSCSI is that it supports only storage devices. USB/IP has the advantage that all types of devices, including isochronous devices, can be controlled over IP networks.

University of Southern California's Netstation [6] is a heterogeneous distributed system composed of processor nodes and network-attached peripherals. The peripherals (e.g., camera, display, emulated disk, etc.) are directly attached to a shared 640Mbps Myrinet or to a 100Mbps Ethernet. The goal of Netstation is to share resources and improve system configuration flexibility. In this system, VISA (Virtual Internet SCSI Adapter) [11] emulates disk drives using UDP/IP. This project is similar to our peripheral bus extension, as both systems use IP network to transfer data [7]. However, while Netstation is a network-based computer architecture that allows easy substitution of systems, our architecture aims to share already-attached devices between heterogeneous computers. We believe it offers the most practical approach to exploiting today's sophisticated peripheral buses and their device drivers.

The Inside Out Network's AnywhereUSB [8] is a network-enabled USB hub. This hub employs proprietary USB over IP technology, and provides remote access to USB devices attached to ports on the hub, though in a somewhat limited manner. The hub supports only USB Bulk and Interrupt devices operating at 12Mbps in a LAN environment. Most USB storage devices, which operate at 480Mbps, and USB isochronous devices are not supported. In contrast, USB/IP supports all types of USB devices operating at up to 480Mbps. Our evaluation has shown that, in LAN environments, all the tested devices work perfectly with the original PDD. Moreover, we have shown that there is an optimization strategy that will enable USB/IP to operate effectively even under larger network delays.

There are a number of network appliances which export the resources of specific USB devices to an IP network. Some NAS appliances share their attached USB storages via NFS or CIFS. In addition, some home routers have USB ports that can be used to share connected USB printers or USB webcams. As we have described in Section 2.1, these appliances, which employ coarse-grained protocols, do not support the low-level operations which a required to allow the remote devices to operate in a transparent manner.

A new technology which is under development is Wireless USB [16], which employs UWB (Ultra Wide Band) to provide expanded USB connectivity. This technology aims to eliminate USB cables altogether, however the effective communication range is limited to 10 meters. The implementation of Wireless USB is very similar to the physical layer of USB, so this technology will complement USB/IP. This will allow USB/IP to be used for Wireless USB devices, and enabling USB/IP to provide remote device access with virtually any IP network infrastructure.

7 Conclusion

We have developed a peripheral bus extension over IP networks, that provides an advanced device sharing architecture for the support of recent sophisticated peripheral bus interfaces. The device sharing architecture meets a range of functionality requirements, including network transparency, interoperability, and generality, by utilizing the low-level device control protocols of the peripheral interfaces.

As a practical example of the peripheral bus extension, we designed and implemented USB/IP, which allows a range of remote USB devices to be used from existing applications without any modification of the application or device drivers. We also undertook a range of experiments to establish that the I/O performance of remote USB devices connected using USB/IP is sufficient for actual usage. We also determined the performance characteristics of USB/IP, and developed optimization criteria for IP networks.

There are three primary design criteria that need to be considered in order to effectively support the transfer of fine-grained device control operations over IP networks. First, for asynchronous devices (known as bulk devices), the device driver must queue enough request data to ensure maximum throughput for remote devices. Second, synchronous devices (known as isochronous devices), require a smooth of I/O stream. To achieve this, the appropriate number of requests must be queued to avoid starvation of requests at the physical device. However, this is a trade-off, because a large queue size reduces the response of each device. Finally, most situations require the response to a request to arrive within a certain time. In some situations, it is possible to relax the restriction by modifying a device driver or an application.

8 Availability

The USB/IP implementation is available under the open source license GPL. The information is at `http://usbip.naist.jp/`.

References

[1] Mark Carson and Darrin Santay. NIST Net: A Linux-Based Network Emulation Tool. *ACM SIGCOMM Computer Communication Review*, 33(3):111–126, 2003.

[2] Stuart Cheshire and Marc Krochmal. DNS-Based Service Discovery. Internet Draft, draft-cheshire-dnsext-dns-sd.txt, Feb 2004.

[3] Stuart Cheshire and Marc Krochmal. Performing DNS Queries via IP Multicast. Internet Draft, draft-cheshire-dnsext-multicastdns.txt, Feb 2004.

[4] Russell Coker. Bonnie++. `http://www.coker.com.au/bonnie++/`.

[5] Cypress Semiconductor Corporation. EZ-USB FX2. `http://www.cypress.com/`.

[6] Gregory G. Finn and Paul Mockapetris. Netstation Architecture: Multi-Gigabit Workstation Network Fabric. In *Proceedings of Interop Engineers' Conference*, 1994.

[7] Steve Hotz, Rodney Van Meter, and Gregory G. Finn. Internet Protocols for Network-Attached Peripherals. In *Proceedings of the Sixth NASA Goddard Conference on Mass Storage Systems and Technologies in Cooperation with Fifteenth IEEE Symposium on Mass Storage Systems*, Mar 1998.

[8] Inside Out Networks. AnywhereUSB. `http://www.ionetworks.com/`.

[9] Stephen Kent and Randall Atkinson. Security Architecture for the Internet Protocol. RFC2406, Nov 1998.

[10] Greg Kroah-Hartman. usbview. `http://sf.net/projects/usbview/`.

[11] Rodney Van Meter, Gregory G. Finn, and Steve Hotz. VISA: Netstation's Virtual Internet SCSI Adapter. *ACM SIGOPS Operating System Review*, 32(5):71–80, Dec 1998.

[12] Tristan Richardson, Quentin Stafford-Fraser, Kenneth R. Wood, and Andy Hopper. Virtual Network Computing. *IEEE Internet Computing*, 2(1):33–38, 1998.

[13] Andrew P. Rifkin, Michael P. Forbes, Richard L. Hamilton, Michael Sabrio, Suryakanta Shah, and Kang Yueh. RFS Architectural Overview. In *USENIX Conference Proceedings*, pages 248–259, Jun 1989.

[14] Julian Satran, Kalman Meth, Costa Sapuntzakis, Mallikarjun Chadalapaka, and Efri Zeidner. Internet Small Computer Systems Interface (iSCSI). RFC3720, Apr 2004.

[15] Sun Microsystems, Inc. NFS: Network File System Protocol Specification. RFC1094, Mar 1989.

[16] Wireless USB Promoter Group. `http://www.usb.org/wusb/home/`.