USB/IP: A Transparent Device Sharing Technology over IP Network

Takahiro Hirofuchi Eiji Kawai Kazutoshi Fujikawa Hideki Sunahara

Nara Institute of Science and Technology

Abstract

Personal computing with affordable computers and their peripheral devices becomes more popular. To use such devices more efficiently and improve their usability, people want to share surrounding peripheral devices between computers without any modification of their own computing environments. The recent device sharing technologies in the pervasive computing area are not sufficient for the peripheral devices of personal computers because these technologies do not provide the network-transparency for applications and device drivers. In this paper, we propose USB/IP as a transparent device sharing mechanism over IP network. This advanced device sharing mechanism is based on the modern sophisticated peripheral interfaces and their supports in operating systems. By the *Virtual Host Controller Interface Driver* we have implemented as a peripheral bus driver, users can share diverse devices over networks without any modification in existing operating systems and applications. The experiments show that USB/IP has fairly practical I/O performance for various USB devices, including isochronous ones. We also describe the performance optimization criteria for the further improvements.

1 Introduction

By the innovations in computing technology, people have their own computing environment with several personal computers. Users always want to use surrounding peripheral devices on-demand and seamlessly through their own computers and their favorite applications, even if these devices are already attached to other computers. For example, a user who brings back his mobile computer to his office may want to make the backup of his working files directly into a DVD-R drive of a shared computer at the office, rather than directly use the shared computer or reconnect the DVD-R drive to his computer. At his desk, he may also wish to work on his mobile computer with an ergonomic keyboard and a mouse which are already-attached to a desktop computer without plugging to a KVM switch. In the context of resource management, the key technology for these scenarios is the network-transparent device sharing mechanism by which computers can interact seamlessly with other computers' devices as well as directly-attached ones.

Many device sharing technologies [2, 3] 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, the network transparency for existing device access interfaces, by which existing applications can also access to remote shared devices without any modification, has not been addressed adequately.

In this paper, we propose USB/IP as a transparent device sharing mechanism over IP (Internet Protocol) network. The main component of this extension is the Virtual Host Controller Interface driver implemented as a peripheral bus driver. It is built in the lowest layer in operating systems so that most applications and device drivers can access remote shared devices through existing interfaces after these devices are *virtually* attached to a computer. We believe that this approach is fairly practical because of recently emerging sophisticated peripheral interfaces and broadband networks.

Our device sharing approach presents several advantages over the conventional approaches. First, a computer can access the full functionality of a shared device. The control granularity of the shared device is the same as directly-attached one. In our system, all the low-level I/O data for the devices are encapsulated into IP packets and then transmitted. Second, a computer can access a shared device with its operating system and applications. With only a few additional device drivers, users can control the

[*] The original paper [1] was published in the Proceedings of the FREENIX Track: USENIX Annual Technical Conference (The USENIX Association: Berkeley, CA, April 2005). Some parts of the FREENIX paper are refined for readers' convenience. Especially, Section 2, Section 3, and Section 5 of the original paper are reconstructed.

shared device as if it was directly attached to a local peripheral bus. Last, 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 structures of operating systems.

In the remainder of this paper, we expand on our vision of USB/IP. Section 2 explains the overview of USB/IP and then gives the design details. Section 3 shows the evaluation of USB/IP and clarifies its characteristics. Section 4 examines related work. We conclude our study in Section 5. The availability of USB/IP is noted in Section 6. The contribution of this paper is that we show the key idea of USB/IP is enough practical by implementing USB/IP on Linux and evaluating its I/O performance.

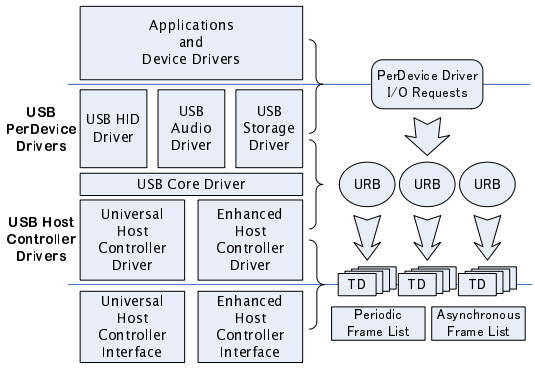

Figure 1: USB Device Driver Model

2 USB/IP

2.1 USB Driver Architecture

USB (Universal Serial Bus) [4] is one of the sophisticated peripheral interfaces based on the recent hardware progress. In the USB 2.0 specification announced in April 2000, a host computer controls various USB devices with 3 transfer speeds (1.5Mbps, 12.0Mbps, and 480Mbps) and 4 transfer types (Control, Bulk, Interrupt, and Isochronous). These transfers are serialized and controlled by a dedicated hardware function which is named USB Host Controller. Figure 1 shows the USB device driver model in most operating systems. A USB Host Controller Driver (USB HCD) exists in the lowest layer of the device driver stack and abstracts the I/O interface of a host controller into a common API for USB device drivers. A USB PerDevice Drivers (USB PDD) is responsible for the control of each USB device. The key design of the USB driver stack is that a USB Host Controller and its USB HCD provide USB PDDs with abstracted data input/output for I/O buffers. Although a USB device and its USB PDD use the buffer to transfer some control data and corresponding reply data, the USB HCD do not distinguish between control and reply.

In USB device drivers, a USB Request Block (URB) presents USB I/O in a controller-independent form, which includes information about I/O;

- I/O buffer
- I/O direction (Input/Output)
- I/O speed (1.5Mbps/12Mbps/480Mbps)
- I/O type (Control/Bulk/ Interrupt/Isochronous)
- I/O destination address
- completion handler

An application or a device driver controls a USB device as follows:

1. A USB PDD converts I/O requests from another driver into URBs and submits these URBs to a USB HCD.
2. The USB HCD transfers data as described by the URB.
3. After I/O of the URB is completed, the completion handler of the URB is called in an interrupt context.
4. The USB PDD notifies the upper driver of the requested I/O completion.

A USB device and its USB PDD use 4 different transfer types depending on characteristics of the device. Control and Bulk transfer types are asynchronously scheduled into the rest of the bandwidth after the periodical transfers. Control transfer is the most fundamental one for enumeration and initialization of devices. In 480Mbps mode, 20% of the bandwidth is reserved for Control transfer. Bulk transfer is used for the requests without any temporal restriction, such as storage device I/O, which is the fastest transfer when the bus is available. Isochronous and Interrupt transfer types are periodically scheduled. Isochronous transfer can move control data at a constant bit rate, which is useful to read image data from a USB camera or to write sound data to a USB speaker. Interrupt transfer confirms the maximum delay of the requested I/O. This is used for USB mice and keyboards, which move a small amount of data sporadically.

2.2 Overview of USB/IP

To share various USB devices between computers, we propose the *Virtual Host Controller Interface* driver as

one of the USB HCDs. The VHCI driver provides the same access interface to a virtually-attached remote USB device. It encapsulates USB requests into IP packets and transmits them to the remote device. This approach makes the best use of attractive USB features, such as various device supports and dynamic device configuration. Furthermore, in the context of device sharing architecture, the advantages of USB/IP are summarized as follows:

Full Functionality. All the functions of remote devices can be manipulated by the operating system. The abstraction on the layer of USB HCDs conceals only the difference between USB host controllers. Per-device operations are fully transferred to remote USB devices.

Network Transparency. The VHCI driver conceals the implementation details of network sharing mechanisms. Other device drivers (e.g., file system, block I/O and virtual memory) and applications do not recognize any difference of the access interfaces between remote devices and locally-attached ones.

Interoperability. If the VHCI driver is implemented in other operating systems, heterogeneous computers can share their USB devices. The USB protocols, which are defined by the several standards, do not depend on the structures in an operating system. Additionally, most operating systems have quite similar USB driver models. Basically, USB/IP is fairly applicable to various operating systems.

To transfer USB I/O data over IP network efficiently, in which transfer delay and jitter are larger than the USB wire, the key point is that USB/IP encapsulates URBs into IP packets. In general, the fine-grained I/O operations with small data size and short temporal restriction depress the total I/O performance when each operation spends more execution time. The native I/O granularity of USB is too small to control USB devices over IP network. Isochronous transfer needs to move 3KB data in every microframe (125us) constantly and Bulk transfer needs to move 6.5KB data in a microframe. The I/O in a microframe is named as "transaction", which is given in Transaction Descriptor (TD). However, a URB, which substitutes a series of USB I/O transactions with one request by concatenating them into its I/O buffer, can relax the restrictions to be ease to handle small I/Os. For example, if a URB represents 80 I/O transactions of Isochronous transfer and each transaction is executed every microframe, the temporal restriction of the I/O is relaxed to 10ms, still keeping its I/O granularity 125us. If a URB has the 100KB buffer of Bulk transport and each transaction moves 512B, this URB substitutes for 200 I/O transactions.

Considering the bandwidth of Gigabit Ethernet goes beyond that of 480Mbps of USB 2.0, it is possible to apply the URB-based I/O model to control remote USB devices over IP network. The evaluation of using URBs for USB/IP is described in Section 3.

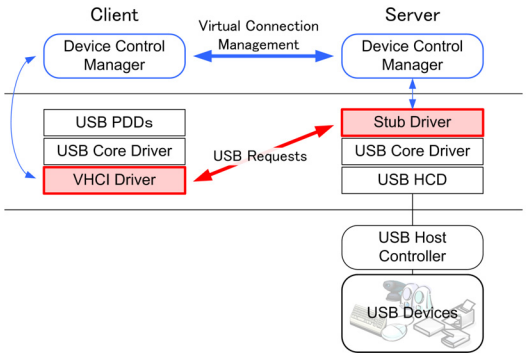

Figure 2: USB/IP Design

2.3 Design and Implementation

The design of USB/IP is illustrated in Figure 2. We create the VHCI driver as a USB HCD in a client host and the Stub driver as a USB PDD in a server host. The VHCI driver emulates the USB Root Hub's behavior; when a remote USB device is connected to a client host over IP network, the VHCI driver notifies the USB Core Driver of the port status change. The Stub driver is responsible for exporting of shared devices. It is automatically loaded when the devices are attached to the server computer. Most URBs are encapsulated/decapsulated into/from IP packets while their device numbers are always converted between the client's one and the server's one.

The USB/IP implementation on Linux uses a common API of a USB core driver in both client and server 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()` translates a URB into a SUBMIT PDU (Protocol Data Unit) (Table 1) 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. It also noted that the USB driver stack and the USB/IP implementation allow a USB PDD to enqueue multiple

Table 1: USB/IP Protocol Data Unit (SUBMIT and RETURN)

Byte	Field
0-3	SUBMIT
4-7	bus number
8-11	device number
12-15	sequence number
16-19	IN/OUT & I/O type & endpoint
20-23	transfer flags
24-27	buffer length
28-31	number of included transaction
32-35	transaction interval
36-39	control buffer
40-	isochronous descriptors (if available)
	I/O buffer (if available)

Byte	Field
0-3	RETURN
4-7	bus number
8-11	device number
12-15	sequence number
16-19	reserved
20-23	transfer flags
24-27	buffer length
28-31	number of included transaction
32-35	error count
36-39	transfer status
40-	isochronous descriptors (if available)
	I/O buffer (if available)

URBs simultaneously for I/O performance improvement.

An existing RPC mechanism, such as Sun RPC, can be used to transfer a URB. However, our prototype implementation simply transfers a URB by defining SUBMIT and RETURN PDUs. This implementation approach is quite simple and practical enough to study the first feasibility of USB/IP.

All PDUs for a virtually-attached USB device are transferred by a TCP/IP connection. In the prototype implementation, the TCP/IP connection is established by userland software and its socket descriptor is passed to the VHCI and Stub driver via `/proc` file system interfaces. To transmit the TCP/IP packets as soon as possible avoiding the buffering delay, the Nagle algorithm [5] [1] is disabled. The current USB/IP implementation does not use UDP/IP communication. The characteristics of the transmission errors of 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. These transaction failures seldom occur unless broken devices or cables are used. Therefore, the transport layer for URBs must guarantee the data arrival in order and retransmit lost packets.

To use remote devices over IP network, the appropriate error recovery should be considered. The error recovery of USB/IP exploits the semantics of USB. If the TCP/IP connection is suddenly disconnected, the VHCI driver detaches the device virtually and the Stub driver resets the device. In the same way as directly-attached USB devices, some applications and drivers may lose their data. However, this recovery policy is appropriate to LAN environments, in which sudden disconnection of TCP/IP seldom occurs. The USB/IP implementation is kept simple by this policy.

[1] This algorithm states that no small packets will be sent on the connection until the existing outstanding data is acknowledged.

```
% 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 3: USB/IP Application Usage

The current 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. Even though shared device access is exclusive, USB/IP gives a fair degree of transparency for remote shared devices.

We have implemented USB/IP for Linux Kernel 2.6 series. A simple USB/IP application is also developed as in Figure 3. As this figure shows, a user selects a shared USB

Table 2: 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

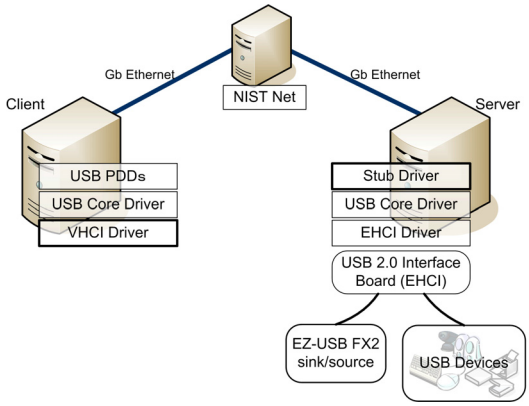

Figure 4: Experiment Environment

device on a server machine and attaches it to a client computer. To attach a remote shared device virtually, a user-land program `devconfig` in the client initiates a TCP/IP connection and passes an established TCP/IP socket to the VHCI driver via `/proc` file system. In this paper, we focus on the basic architecture of USB/IP and its performance evaluation. The device discovery, the authentication and the security will be described in another paper.

3 Evaluation

In this section, we show the USB/IP characteristics. We have conducted several experiments to measure the USB/IP performance. The used computers are listed in Table 2. To emulate various network conditions, we use the NIST Net [6] package on Linux Kernel 2.4.18. A client machine and a server machine are connected via a NIST Net machine, shown in Figure 4. Both the client and server machines run Linux Kernel 2.6.8 with the USB/IP kernel modules.

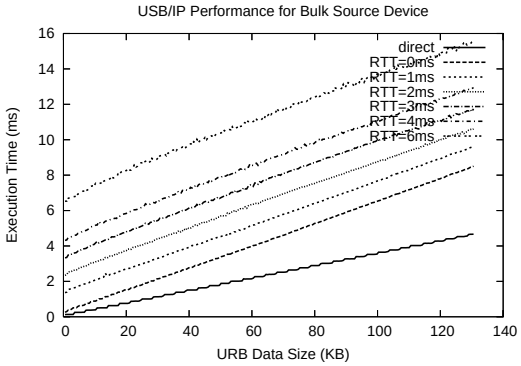

Figure 6: Execution Time of a URB

3.1 Performance Evaluation of Data Sink/Source

In this subsection, we used a pure sink/source USB device rather than a consumer USB device, to clarify the characteristics of USB/IP itself. A USB peripheral development board with a Cypress Semiconductor EZ-USB FX2 chip [7] was programmed to be a sink/source of USB data. When programmed as a data sink, the board receives output data of Bulk or Isochronous transfer types from a computer. When programmed as a data source, the board sends input data of Bulk or Isochronous transfer types to a computer. We wrote its firmwares and the test device driver as a USB PDD for the experiments.

3.1.1 Bulk Transfer

The I/O performance of USB/IP depends on network delay and the data size of the operation. In this subsection, we derive the modeled throughput from the results of experiments and give the optimization criteria.

The USB/IP overhead for USB requests of Bulk transfer was measured. As Figure 5 shows, the test driver submits a Bulk URB to the remote source USB device, waits for its completion and then resubmits it continuously. Each execution time of URB in the client was measured by the TSC register of Intel Pentium processors, changing URB data size and network RTT. When NIST Net sets network RTT to 0ms, the actual RTT between a client machine and a server machine is 0.12ms by `ping`.

The result is shown in Figure 6. The execution times of URBs are linear to the data size and the gradients in each case are constant. The CPU cost for the USB/IP encapsulation is quite low; in these experiments, it is always under a few percentages. There is no influence of the TCP/IP buffering because `TCP_NODELAY` is set in the socket options. From the graph, the execution times t_{overIP} for the

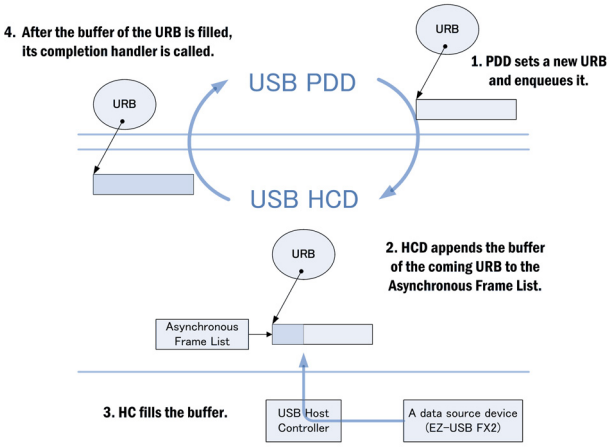

Figure 5: The Summarized Driver Behavior of the Experiments in Section 3.1.1. In the case of USB/IP, the enqueued URBs are transferred to the server's HCD between 1 and 2, and vice versa between 3 and 4.

data size s is

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

a_{overIP} is the gradient value in the USB/IP cases. Then the throughput *thpt* is

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

The regression analysis shows a_{overIP} is 6.30e-2 ms/KB and the intercept in the 0ms case is 0.24 ms. The modeled throughput by Equation (Figure 1) is illustrated in Figure 7. The actual throughput from the experiments is illustrated in Figure 8. As these graphs show the same shape, the throughput of the model is fully substantiated by the experimental results. Within the parameter range of the experiments, this model is appropriate to estimate the throughput with URB data size and network RTT.

To summarize these experiments, we have estimated the appropriate URB data size under various network delays. With the additional experiments in which multiple URBs were enqueued simultaneously, we have confirmed that the throughput of Bulk transfer is dependent on the total I/O data size of simultaneously-enqueued URBs. To keep enough throughput with some network delay, USB PDDs should enlarge each URB data size or the queuing depth of URBs. Moreover, when a large amount of URBs are enqueued asynchronously under larger network delay, the TCP/IP window size must be also enlarged to fill up the network pipe.

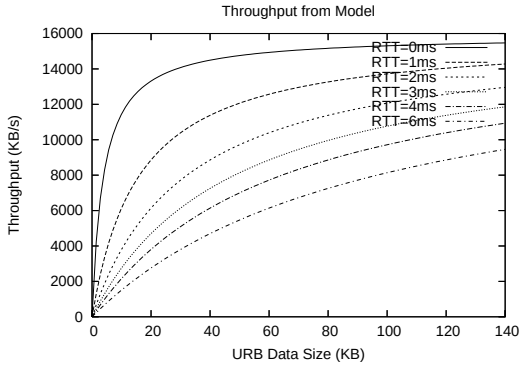

Figure 7: Throughput from Model

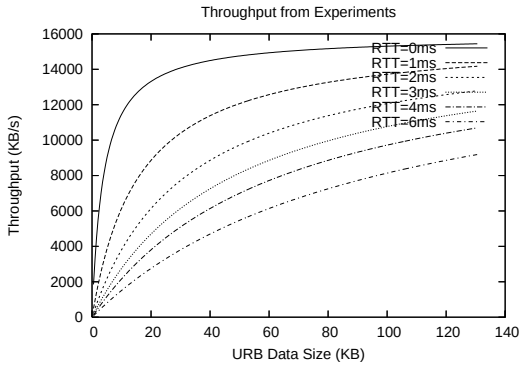

Figure 8: Throughput from Experiments

3.1.2 Isochronous Transfer

In this subsection, we examine the isochrony of USB/IP. To keep periodical transfers for USB Isochronous devices,

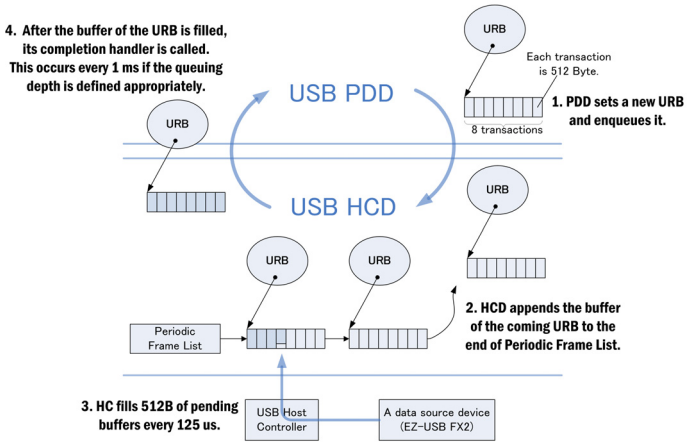

Figure 9: The Summarized Driver Behavior of the Experiments in Section 3.1.2. In the case of USB/IP, the enqueued URBs are transferred to the server's HCD between 1 and 2, and vice versa between 3 and 4.

the starvation of transaction requests must be avoided in the host controller. Therefore, the USB driver model allows USB PDDs to enqueue multiple URBs simultaneously. In the case of USB/IP, it is important to select the appropriate queuing depth of URB, as determined by network delays.

We wrote the firmwares and the 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 ($125us \times 8$) [2]. Figure 9 shows the detail of the driver behavior in the experiments. The USB PDD sets up each URB with the pointer of I/O buffer for 8 transactions and then enqueues multiple URBs. As a result, while the host controller keeps pending I/O requests in the Periodic Frame List, the completion handler is called every 1ms and the HCD moves the input data to the USB PDD periodically. On the other hand, if there is no pending I/O request in the Periodic Frame List, the host controller does not copy data and isochronous data will be lost.

For the directly-attached source device, when the USB PDD submitted only one URB, the completion intervals became 11.1ms because of the starvation of the requests.

[2]The 1ms interval in the experiments is selected to become enough small to examine the isochrony of USB/IP. In general, the interval of completion is defined by driver developers as approximately 10ms considering the smoothness of I/O and the processing cost.

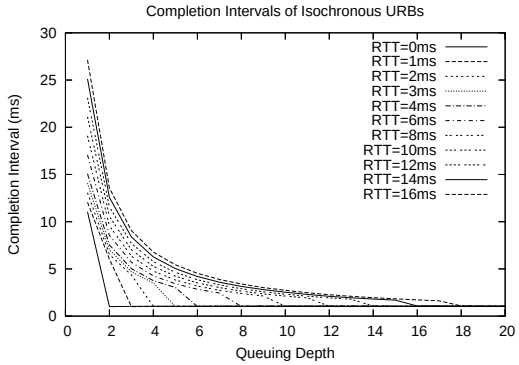

Figure 10: Mean Completion Intervals

When the USB PDD submitted 2 or more URBs simultaneously, the completion intervals kept 1ms constantly. The standard deviations were approximately equal to 20 ns for any queuing depth.

In the case of USB/IP, Figure 10 illustrates the mean completion intervals for various network RTTs and the queuing depths of submitted URBs. Even under some network delays, the USB PDD could get 1ms completion intervals with the enough queuing depths. For example, when the network delay is 8ms, the appropriate queuing depth is 10 and more. In this condition, time series of completion intervals are figured in Figure 11. Immediately after the start of the transfer, the completion intervals vary widely because the host controller is not still filled with enough URBs. Once the host controller is filled and the cycle becomes stable, the completion intervals keep

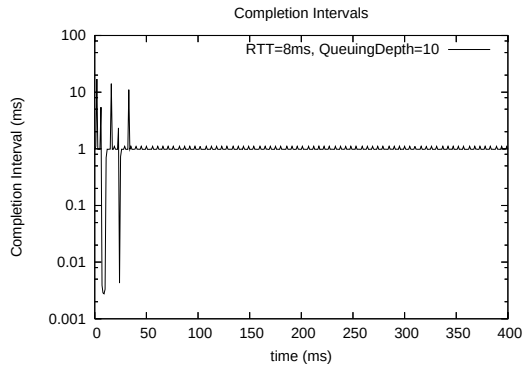

Figure 11: Time Series Data of Completion Intervals (RTT=8ms, Queuing Depth=10)

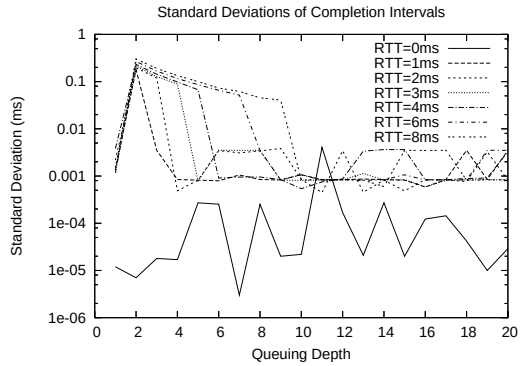

Figure 12: Standard Deviations of Completion Intervals

1ms. Figure 12 shows the standard deviations of completion intervals. With the sufficient URBs, the measured standard deviations are less than 10us, including the NIST Net's deviations [6]. These values are less than one microframe interval (125us) and enough accurate for most device drivers, because process scheduling is driven by `jiffies`, which is incremented every 1ms in Linux kernel 2.6 series. There is no influence of the TCP/IP buffering because of the socket option `TCP_NODELAY`.

The periodical transfers of USB/IP are illustrated in Figure 13. In this case, the USB PDD in the client host enqueues 3 URBs which are completed every x ms. For periodical completions in the server host, the host controller must always keep multiple URBs enqueued. Therefore, with the queuing depth q, the time in which the next URB is pending is

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

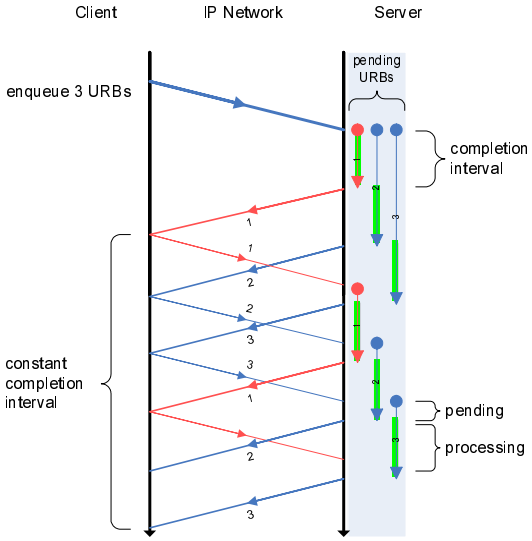

Figure 13: USB/IP Model for Periodical Transfers

Then, the appropriate queuing depth q is calculated by

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

In Figure 10, Equation (2) can explain the sufficient queuing depth of URBs in the experiments.

To summarize these experiments, USB PDDs with periodic transfers must enqueue multiple URBs to fill the queuing depth q of Equation (2) for continuous stream I/O. In the other networks rather than LAN, with jitter or packet loss, q should be increased with sufficient margins. The result can be also applied to Interrupt URBs which specify the maximum delay of completion. This examination continues for common USB devices over IP network in Section 3.2.2.

3.2 Performance Evaluation of USB Devices

In this subsection, we show the USB/IP characteristics for common USB devices. All USB devices we have tested can be used as USB/IP devices. Figure 14 shows that a client host virtually attaches a remote USB camera through the VHCI driver and a USB device viewer `usbview` [8] gets the device descriptors. No difference between USB and USB/IP can be seen except that the host controller is VHCI. USB PDDs can also control their corresponding remote USB devices without any modification themselves. In our LAN environment, the performance degradation of USB/IP is negligible for actual usages. The

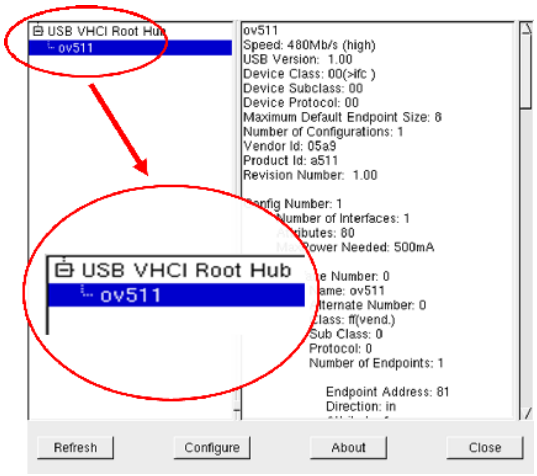

Figure 14: `usbview` Output for a USB/IP Device

Table 3: Specification 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

specific details of each kind of USB/IP devices are described in the next paragraphs. It should be noted that the performance of USB/IP becomes worse under a congestion network. However, this issue will be addressed in future work. For example, to transfer USB requests beyond wide area network, a reservation mechanism of network resources may be employed with USB/IP.

3.2.1 USB Bulk Device

USB storage devices (e.g., hard disks, DVD-ROM drives, memory drives, etc.) , USB printers, USB scanners and USB Ethernet devices use USB Bulk transfer mainly. We can use these devices by USB/IP; for USB storage devices, we can make partitions, file systems, mount/umount and operate files. Moreover, we can play DVD videos in remote DVD drives and can also write DVD-R media, with existing applications. As described in Section 3.1.1, 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 experiment environments are the same as those described in Section 3.1.1. NIST Net was used to emulate various network delays. We have run Bonnie++ 1.03 [9] benchmarks for ext3 file system on a USB hard disk described in Table 3. Bonnie++ measures performances of hard drives and file systems, by the file I/O tests and the file creation/deletion tests. The file I/O tests include sequential I/O per character and per block and random seeks. The file creation/deletion tests do `creat/stat/unlink` a lot of small files.

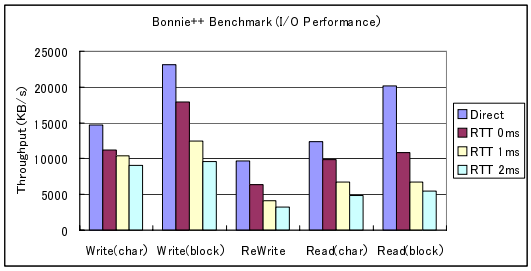

Figure 15: Bonnie++ Benchmark (Sequential Read/Write Throughput on USB and USB/IP)

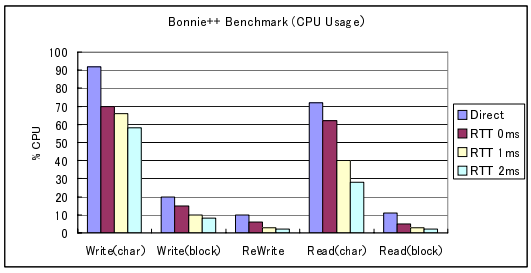

Figure 16: Bonnie++ Benchmark (Sequential Read/Write CPU Usage on USB and USB/IP)

Figure 15 and Figure 16 show the sequential I/O throughput and their CPU usages on the client by USB and USB/IP, respectively. Figure 17 shows sequential

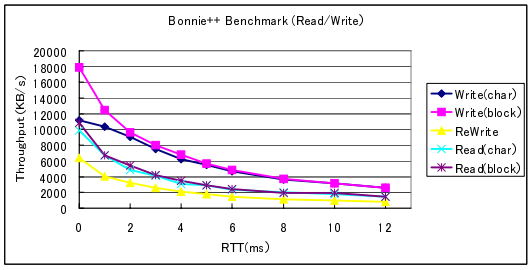

Figure 17: Bonnie++ Benchmark (Sequential Read/Write Throughput on USB/IP)

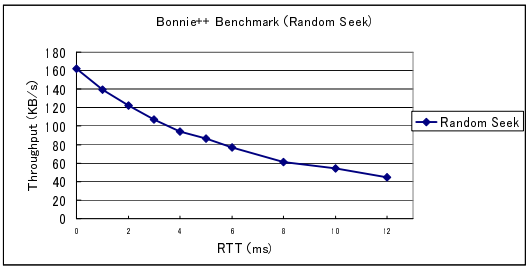

Figure 18: Bonnie++ Benchmark (Random Seek Speed on USB/IP)

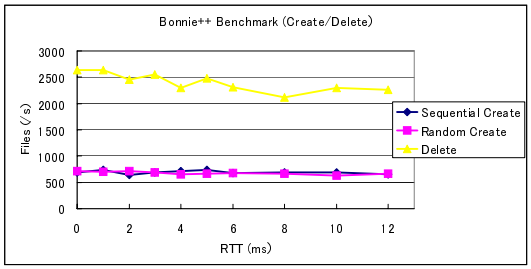

Figure 19: Bonnie++ Benchmark (Create/Delete Speed on USB/IP)

I/O throughput by USB/IP under various network delays. For char write and block write, the throughput by USB/IP when NIST Net's RTT is 0ms (0.12ms by `ping`) is approximately 77% of the throughput by only USB, for rewrite 66%, for char read and block read 79% and 54%, respectively. Because the graph shows that there is enough room in the CPU usage, the performance degradation of the USB/IP case can be attributed to insufficient queuing data size. 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, a total I/O data size of URBs submitted simultaneously is the same as each SCSI request size; in the case of the block write, this is approximately 128KB. This queuing data size is small for USB/IP under some network delays, as we have considered in Section 3.1.1. To optimize the sequential I/O throughput for USB/IP, the reasonable solution is that the USB storage driver should provide the SCSI request queuing.

The throughput of random seek I/O by USB/IP is illustrated in Figure 18. This test runs a total of 8000 `lseek` in a file randomly with three seek processes. A seek process reads a block in each case and also writes back the block in 10% of cases. The throughput by only USB is 167KB/s. The throughput difference between USB and USB/IP is quite smaller than that of sequential I/Os. The CPU usages in both the USB and USB/IP cases are 0%. Random seek I/O has its bottleneck in the seek speed of a USB hard disk itself. The seek speed of the hard disk is relatively slower than those of read/write I/Os; the rational speed of USB hard disk tested is 5400rpm and its seek speed is approximately 10ms.

Figure 19 shows the speed of file creation and deletion by USB/IP under various network delays. The speeds by USB are 682/s in the sequential creation, 719/s in the random creation and 2687/s in the deletion. In all the cases, the CPU usages are over 90%. The differences between USB and USB/IP are quite small and the speeds of each test are almost constant under various network delays. In the file creation/deletion tests, the bottleneck is the CPU resources.

3.2.2 USB Isochronous Device

USB multimedia devices, such as USB cameras and USB speakers, use USB Isochronous transfer to move their periodical data. We have tested a USB camera with the OmniVison OV511 chip and a USB Audio Class speaker. These devices work completely by USB/IP in our LAN; we can capture the video from the camera and play some music by the speaker.

The Linux USB Audio Class driver submits 2 URBs simultaneously for multi-buffering. Each URB is responsible for 5ms transactions. Equation (2) shows this driver can play the speaker by USB/IP under less than 5ms network delays. For larger network delays, the easy and effective solution is that the completion interval of each URB should be enlarged. However, there is also the drawback that such modification makes the I/O response worse than ever.

3.2.3 USB Interrupt Device

USB Human Input Devices, such as USB keyboards and USB mice, use USB Interrupt transfer to move data sporadically like IRQ. Some other devices also use the Interrupt transfer to notify their status changes. In our LAN, we can use these devices by USB/IP comfortably with consoles and X Window System.

Most USB HID drivers submit only one URB whose completion delay is 10ms. After the URB is completed, the driver resubmits it. The drivers read the interrupt data every 10ms, which is accumulated in the device endpoint

buffer. Under large network delays, the possible pitfall is that the device endpoint buffer may overflow. Under more than 100ms network delays, the users may feel odd for their input devices. The former problem can be resolved by enqueuing more URBs not to overflow the endpoint buffer. The latter problem is underlying in any network program with human interaction.

4 Related Work

iSCSI [10] transports the SCSI packets over TCP/IP and provides the access to storage devices beyond IP networks. It is worthy of remark that iSCSI is the extension of a SCSI bus to IP networks and its protocol has basically network transparency and interoperability. However, iSCSI supports only storage devices. Our USB/IP has the advantage that all types of devices, including isochronous devices, can be controlled over IP networks.

Network File System (NFS) [11] is widely used to share storage devices between computers. 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. However, USB/IP provides all the functions of a remote shared device because it is implemented in the lowest layer of an operating system.

University of Southern California's Netstation [12, 13, 14] 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) [15] emulates disk drives using UDP/IP. This project is similar to our proposed system; both systems use IP network to transfer peripherals' data [16]. However, while Netstation studied network-based computer architecture and substitutes existing systems, our system aims to share already-attached devices between heterogeneous computers and proposes the most practical approach in existing operating systems and applications with exploiting today's sophisticated peripheral buses and their device drivers.

The Inside Out Network's AnywhereUSB [17] is a network-enabled USB hub. By their proprietary USB over IP technology, it provides remote access to the USB devices attached to its USB ports in a quite limited manner. It supports only USB Bulk and Interrupt devices within 12Mbps mode under LAN environments; most USB storage devices, which are now implemented as 480 Mbps devices, and USB isochronous devices are out of its target. Our USB/IP supports all types of USB devices with 480Mbps mode. Our evaluation has shown that in LAN environments all the devices work perfectly with the original PDD. Moreover, the optimization strategy is also presented to utilize our USB/IP more effectively even under larger network delays.

As one of the applications of USB/IP, it is possible to develop device switching software like existing KVM (Keyboard, Video, and Mouse) switches. These switching appliances have a virtual USB keyboard and mouse inside to always emulate these attachments to computers. This approach is also appropriate to the device switching software of USB/IP. USB/IP can provide all kinds of USB devices with fairly direct access over IP networks, which are now deployed everywhere by diverse media. This advantage has enormous potentialities for useful applications. A practical application of the USB/IP technology is now in progress to share a device more easily between computers.

Wireless USB [18], which is under development, employs UWB (Ultra Wide Band) to expand the USB connectivity. This technology aims to eliminate USB cables. The communication range is limited within 10 meters. The implementation of Wireless USB is almost in the physical layer of USB. Therefore, this technology is a complement for our USB/IP; our USB/IP will be also used for Wireless USB devices. Our USB/IP provides remote device access with IP network infrastructures which are deployed everywhere.

5 Conclusion

We designed and implemented USB/IP as a transparent device sharing mechanism. We can use various remote USB devices from existing applications without any modification of the device drivers. In our LAN, the I/O performance of remote USB devices is sufficient for actual usage. By the experiments, we showed the characteristics of USB/IP and the optimization criteria for IP networks.

To transfer fine-grained device control operations over IP networks, three design criteria should be considered carefully for the applied networks. First, for asynchronous devices as known as bulk devices, the device drivers should enqueue enough request data to get the maximum throughput for remote devices. Second, for synchronous devices as known as isochronous devices, the smoothness of I/O is required. The appropriate number of requests should be enqueued to avoid the starvation of requests on the device side. However, large queuing size makes the

responses of each device worse. Last, the temporal restrictions of the request responses should be considered. In greater or lesser degrees, all the requests have their own temporal restrictions of the responses. As far as the system works properly, it is possible to relax the temporal restrictions in the device drivers or the applications and also possible to keep the buffering size minimum in synchronous transfers.

In future work, we continue to improve the USB/IP technology to support various network environments efficiently, such as a wireless network and a WAN.

6 Availability

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

References

[1] Takahiro Hirofuchi, Eiji Kawai, Kazutoshi Fujikawa, and Hideki Sunahara. Usb/ip - a peripheral bus extension for device sharing over ip network. In *Proceedings of the FREENIX Track: 2005 USENIX Annual Technical Conference*, Apr 2005.

[2] Universal Plug and Play. `http://www.upnp.org`.

[3] Jini. `http://www.jini.org`.

[4] USB Implementation Forum, Inc. `http://www.usb.org/`.

[5] John Nagle. Congestion control in TCP/IP internetworks. `http://www.ietf.org/rfc/rfc896.txt`, Jan 1984.

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

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

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

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

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

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

[12] Gregory G. Finn. An Integration of Network Communication with Workstation Architecture. *ACM SIGCOMM Computer Communication Review*, 21(5):18–29, 1991.

[13] Gregory G. Finn, Steve Hotz, and Rod Van Meter. NVD Research Issues & Preliminary Model, March 1995.

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

[15] 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.

[16] 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.

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

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