NAIST-IS-DD0461032

Doctoral Dissertation

USB/IP: Universal Serial Bus Extension over IP Network

Takahiro Hirofuchi

February 1, 2007

Department of Information Systems
Graduate School of Information Science
Nara Institute of Science and Technology

A Doctoral Dissertation
submitted to Graduate School of Information Science,
Nara Institute of Science and Technology
in partial fulfillment of the requirements for the degree of
Doctor of ENGINEERING

Takahiro Hirofuchi

Thesis Committee:

Professor Hideki Sunahara	(Supervisor)
Professor Suguru Yamaguchi	(Co-supervisor)
Associate Professor Kazutoshi Fujikawa	(Co-supervisor)
Associate Professor Eiji Kawai	(Co-supervisor)

USB/IP: Universal Serial Bus Extension over IP Network*

Takahiro Hirofuchi

Abstract

In the ideal goal of distributed system technologies, users obtain optimized computing environments dynamically composed of distributed hardware resources over networks on-demand and seamlessly. A remote device technology is one of the essential keys to this goal. It enables a system to be reconfigurable and adaptable in its device usages for various conditions and purposes. Several remote device technologies have been studied for many years. However, there is still lacking a fully general and interoperable device access mechanism over IP networks, though this is considered necessary for tackling the increasing diversity of devices and computers. Most remote device technologies in operating systems involve operating-system-specific abstractions in their mechanisms.

This dissertation proposes a network extension of USB, a multipurpose peripheral interface, via host controller virtualization in operating systems. It decouples device requests and messaging from bus controls, and therefore enables remote access requests and semantics to become the same as physically-attached local devices; these are independent of operating systems and specific to their peripheral bus models. It is conducive to transparent access for various types of remote devices in the same manner as local devices, with interoperability among different operating systems.

However, since its peripheral interface is not intended to be extended over networks, this research addresses inevitable semantics differences and problems arising from its peripheral bus model. It clarifies feasibility and restrictions of the network extension, as well as suggestions for the bus models and implementations.

*Doctoral Dissertation, Department of Information Systems, Graduate School of Information Science, Nara Institute of Science and Technology, NAIST-IS-DD0461032, February 1, 2007.

A network extension of the USB model is discussed in detail through the development and evaluations of USB/IP (USB request over IP). Its design criteria are presented to address performance issues as well as semantics alteration criteria about time constraints, request scheduling, bandwidth, synchronization, and request cancellation. Its I/O performance model is established through its implementation and evaluation, which achieves feasible I/O performance for all USB devices under a LAN environment. A device sharing system is also developed as one of the applications of the proposed mechanism, through which its impact against the device management of operating systems is discussed.

Keywords:

Remote Device, Peripheral Bus, IP Network, Operating System, Device Driver, Universal Serial Bus (USB), Host Controller, Virtualization

Contents

1 Introduction **1**

2 Remote Device Technologies **4**

2.1 Architecture Components 5
2.2 Taxonomic Criteria 6
2.3 Related Work 8
2.3.1 Peripheral I/O Bus 8
2.3.2 Network-Attached-Peripherals Computer 11
2.3.3 Operating System 11
2.3.4 Middleware and Application 16
2.4 Summary 17

3 USB/IP **19**

3.1 USB Design Model 20
3.1.1 USB Per-Device Driver 21
3.1.2 USB Core Driver 22
3.1.3 USB Host Controller Driver 23
3.1.4 USB Request Block 23
3.2 Requirement 24
3.2.1 Semantics Alteration 26
3.3 Design 28
3.3.1 Virtual Host Controller Interface Driver 28
3.3.2 Protocol Data Unit Based on USB Request Block 29
3.3.3 Stub Driver 29
3.3.4 Prototype Implementation 30

3.4 Evaluation 32
3.4.1 Performance Evaluation of USB Pure Sink/Source 34
3.4.2 Performance Evaluation of USB Devices 44
3.4.3 Performance Evaluation under IPSec 49
3.5 Discussion 51
3.5.1 Request Semantics 51
3.5.2 Error Recovery 54
3.5.3 Interoperability 55
3.5.4 Dependability 56
3.5.5 Naming 57
3.6 Related Work 57
3.6.1 iSCSI 57
3.6.2 Software and Hardware for USB Device Sharing 58
3.6.3 KVM Switch and NAS 58
3.6.4 Wireless USB 59
3.7 Summary 59

4 A Device Sharing System for USB/IP 61
4.1 Device Sharing Based on USB/IP 61
4.2 Requirements 62
4.2.1 Location-Independent Handling 62
4.2.2 Device Ownership and Control 65
4.2.3 Device Management Redesign 65
4.3 Design 66
4.3.1 Device Presentation 66
4.3.2 Device Management Model 69
4.4 Implementation 73
4.4.1 Strategy 73
4.4.2 Overview 76
4.4.3 Prototype Demonstration 79
4.5 Discussion 80
4.6 Summary 82

5 Open Issue 83

6 Conclusion **85**
6.1 Contribution 85
6.2 Futher Directions 86

Acknowledgments **88**

Availability **89**

References **90**

Appendix **101**

Publications **108**

List of Figures

3.1 USB Design Model 22
3.2 USB/IP Design 28
3.3 VHCI Driver Implementation 31
3.4 USB/IP Application 33
3.5 USB/IP Application Usage 33
3.6 Experiment Environment 34
3.7 Driver Behavior of Asynchronous Transfers 35
3.8 Execution Time of a URB 36
3.9 Throughput from Model 37
3.10 Throughput from Experiments 38
3.11 Driver Behavior of Periodical Transfers 39
3.12 Mean Completion Intervals 41
3.13 Time Series Data of Completion Intervals 41
3.14 Standard Deviations of Completion Intervals 42
3.15 USB/IP Model for Periodical Transfers 43
3.16 Output Example of USB Utility 44
3.17 Sequential Read/Write Throughput 46
3.18 Sequential Read/Write CPU Usage 46
3.19 Sequential Read/Write Throughput (RTT) 46
3.20 Random Seek Speed 47
3.21 Create/Delete Speed 48
3.22 Sequential Read/Write Throughput with IPSec 50
3.23 Sequential Read/Write CPU Usage with IPSec 50

4.1 Device View without USB/IP 63

4.2 Device View with USB/IP 64
4.3 Flat Device View for Device Sharing 64
4.4 Device Object Operations 67
4.5 Device Object and Flat Device View 67
4.6 Device Management Model 71
4.7 Delegation in Device Management 72
4.8 Delegation between Two Computers 73
4.9 Trust Management Concept 74
4.10 Design Overview 77
4.11 System Behavior (Local Request) 78
4.12 System Behavior (Remote Request) 78
4.13 Assertion Example 79
4.14 Usage Example 81

List of Tables

3.1 Machine Specifications for Experiments 34
3.2 Specifications of Tested USB Hard Disk 45

A.1 Assertion Parameters 107

Chapter 1

Introduction

A computer is a machine that stores and processes data according to programs. A device is defined as one of the components of a computer, through which various data is inputted (outputted) to (from) the computer. A device, for instance, means a storage device such as a hard disk drive or an optical drive, a human interaction device such as a keyboard or a mouse, or a multimedia device such as a camera or a speaker. In general, the concept of device access in operating systems covers a wide range of interactions between a computer and a device; in the closest level to hardware it means direct control of device hardware (e.g., register I/O of a disk controller), and in the furthest level it implies resource usages of abstracted functions derived from device hardware.

Remote device access over networks brings efficient utilization of computer resources and advanced usability. When a user can access remote devices over networks, a computer can use other devices over external computer networks rather than use only physically-attached devices through its peripheral interfaces. Ideally, device access is not closed inside the computer, but opened to all the network-accessible devices. A device is seamlessly provided to any computer on demand regardless of its physical location. The physical relationship between a computer and a device is at once decoupled and then dynamically recoupled over networks for on-demand and efficient usages.

The advantages of remote device technologies in distributed systems are summarized as the enhancement of reconfigurability and adaptability in regard to device utilization:

Reconfigurability. The ability to provide computers with device hardware and its derived resources dynamically and flexibly. A system can dynamically reconfigure optimized utilization of devices against various conditions under distributed systems. For example, a computer can access remote devices of another computer anytime without reconnecting them physically, even if the computer moves to another place.

Adaptability. The capability of sustainable adaptation to changes in hardware and software of devices and computers under distributed systems. A system can follow technical changes without unnecessary modifications by an abstraction between a remote device and a computer, since the abstraction decouples their fixed relationship and advances flexibility of coordination. For instance, a file sharing protocol used in a storage area network gives an absorbing layer of differences; even when a storage virtualization mechanism in a server host is changed, a client host can still access the same storage through a common file sharing protocol without any modification on the client side's software.

Many remote device technologies have been studied over the years. However, these technologies do not meet device generality and interoperability among operating systems, though these attributes are considered necessary to address the growing diversity of devices and computers. Actually, most remote device technologies are based on interfaces and semantics of operating systems. Even though they extend device access semantics to be suitable for remote access via networks, these technologies are essentially dependent on device abstraction performed by operating systems. Their interoperability among different operating systems becomes impossible to achieve while fully preserving the same semantics as the local devices.

This dissertation proposes a network extension mechanism of USB, a multipurpose peripheral interface with host controller support, which enables transparent remote access to various types of devices in an interoperable manner among different operating systems. A virtualized host controller decouples request messaging to devices from bus controls, and enables remote device access to exploit the same request messages and semantics as physically-attached local devices. The request messages and semantics used for remote device access are basically not operating-system-dependent but hardware-dependent, which is a straightforward

extension of peripheral bus models defined by specifications.

However, its peripheral bus model is designed for only physically-attached devices; since the semantics of remote access cannot become fully the same as those of local access, the proposed mechanism needs to address problems arising from inevitable differences between local and remote semantics. Through its design and implementation, this research explores the possibility and feasibility of the proposed extension, and also clarifies its limitations and restrictions. It discusses design criteria of the network extension mechanism as well as criticism of peripheral bus models and implementations.

Chapter 2 reviews existing remote device technologies. They suggest hints for an alternative remote device mechanism. In Chapter 3, the network extension mechanism is proposed. USB/IP is designed and implemented as a network extension of the USB model. Its design criteria are discussed to address performance issues as well as semantics alteration criteria (i.e., time constraints, request scheduling, bandwidth, synchronization, and request cancellation). Suggestions for the USB model and implementations are also discussed for suitable support of the network extension. Chapter 4 presents the device sharing system of the proposed mechanism to focus on its differences from other remote device technologies in operating systems. The proposed remote device mechanism never abstracts remote devices and multiplexes device access by itself. It should work seamlessly with existing device access mechanisms to be incorporated into device management of operating systems. Design criteria of its applications are clarified through the development and experiences of the device sharing system. Chapter 5 discusses other multipurpose peripheral interfaces with host controller support, and Chapter 6 concludes this dissertation.

Chapter 2

Remote Device Technologies

Nowadays, the variety of peripherals, computers, and software is dramatically increased in a ubiquitous computing world. In days past, people used to think of a computer's peripheral as only a storage device, a printer, and a terminal console. However, in recent years, various devices have been developed as computer peripherals that interact with the computers, and this trend will continue. A range of computers has also become indispensable in our daily lives, used not only as workstations but also as laptop computers, PDAs, smart phones, and home appliances. Various types of software for these computers have been developed and customized in all the layers from operating system kernels to applications. On the other side of this diversity, the technology that links and coordinates these peripherals, computers, and software is strongly recommended to be de-fact or standardized for interoperability. IP networks are widely deployed and are versatile with network media including wired and wireless links. Some network media are as fast and as broad as peripheral buses.

This thesis first discusses whether existing remote device technologies are sufficient for the coming years. Remote device technologies have been developed for specific purposes reflecting technical backgrounds from the past that are now rapidly changing. It is considered essential that at least one mechanism of remote device access meets the following requirements to address diversity and interoperability in hardware and software.

- The technology can be applied to various types of devices. It should support naturally emerging devices with new functions.

- The technology uses IP network for its transport. It is the most general protocol used for addressing and messaging of remote device access.
- The technology has natural interoperability between various operating systems and computers. It should be able to be a standardized technology.

In this chapter, system components and technical features of technologies are first summarized, thereby presenting technical focuses for categorizing. Then, existing remote device technologies are noted pointing out their design principles. In this thesis, the term *remote device architecture* is used to present the big perspective of technologies and classify them by their design principles.

2.1 Architecture Components

This section describes the major components of remote device architecture. To review software and hardware technologies of remote device architecture, these components are important to clarify its technical background and its purpose. It should be remarked what type of technology is supposed to be a component of it. Basically, these components are common to all kinds of architectures and are implemented newly in a remote device technology or included naturally in a target hardware or software technology.

Computer Hardware and System Software. Remote device technologies are designed for target computer platforms (e.g., personal computers, mobile computers, and cluster computers). Various computer platforms are available supporting remote device access in several ways and have different requirements and purposes for remote devices. Remote device architecture also tends to be deeply dependent on operating system architecture. It is the most fundamental mechanism of resource management on a computer.

Networking Mechanism. A networking mechanism interconnects computers and devices, sometimes transferring remote device I/O messages. It distinguishes remote device technologies. This mechanism covers network media, the network interface of a computer, its device driver and network protocol stacks. A peripheral bus sometimes works for interconnection between computers. Some remote device mechanisms are designed by exploiting a general network protocol

independent of network media, but others are fully-based on the special protocol of network media. The priority management mechanism of network traffic sometimes facilitates efficient remote device I/O under shared networks.

Target Device. A remote device technology is designed for specific types of target devices or not. Moreover, two types of remote devices exist; one type is connected to networks between computers directly, and the other type is first connected to a peripheral bus of a computer and then exported to other computers over networks. Most client computers accessing remote devices do not see any semantic differences between them. In the later case, a server computer needs to have an arbitration mechanism between local access and remote access. In this thesis, simply using *remote device* means both types of devices.

Device I/O Software. Various software mechanisms dedicate device I/O in various system layers; close to hardware (i.e., device driver) or far from hardware (i.e., userland application). Each system layer abstracts device I/O and presents common interfaces to its upper layer. Through the I/O path of remote device access, some system layers are implemented on a remote device side, and others on a client side, thereby transferring abstracted I/O messages over networks. This abstraction scheme dominates the technical features of a remote device technology.

The above components are fully related to the I/O path of remote device access. They are indispensable to take a broad view of remote device architecture. Other system components not related to remote device I/O directly, such as a directory service and an authentication service, are not focused on to make the scope of the discussion clear.

2.2 Taxonomic Criteria

This section describes possible design criteria that characterize remote device architecture.

Functionality. A computer can access remote device functions through an abstraction layer between them. The functionality of a remote device is dependent on the method of abstraction. To take an example of a storage device, a *file* is an abstracted form of a storage device. An application has an interface to a

file via file operations (e.g., `open`, `read`, `write`, and `close`) to store and load data. Block I/O is another abstracted form, which allows random block access to any offset of a physical storage. In general, file abstraction is appropriate to most userland applications, and block I/O abstraction is suitable for file system drivers of operating systems and some database applications.

Sharing Scheme. A remote device is shared by multiple computers or not. The method of device sharing depends on the device abstraction. Some remote device technologies support concurrent access to one device, and others may allow exclusive device sharing in a time-divisional way among computers.

Disconnected Operation. A remote device technology is designed for sudden or intended disconnection of a target device. Some remote device technologies are based on the fact that their target devices are never disconnected. Other technologies may support disconnected operations, just stopping device usages safely or actively allowing the same operations even when disconnected. Aggressive error recovery is sometimes implemented considering its target environment. If an intelligent disconnected operation is indispensable to remote device access, the abstraction of a target device is first carefully designed to support it.

Generality. A remote device technology covers various kinds of devices, or only covers a special class of device. The technology may be designed for a specific type of device to be optimized to its functionality. A more general mechanism is also possible to support diverse devices. The generality is sometimes dependent on the design of its peripheral interface and its device driver model.

Heterogeneity. A remote device technology is applied to various computers and operating systems, or only to a specific computer and its operating system. Since distributed systems are now composed of heterogeneous hardware and software, interoperable technologies are desirable also for remote device access.

Performance. The I/O performance of remote devices should be sufficient for actual usages. It is affected by network media, network conditions, processing power of a remote device and a computer, and the method of device abstraction. Some remote device technologies are supposed to be used under broad bandwidth, and others may work under narrow bandwidth. The caching or pre-fetching mechanisms are sometimes supported to increase I/O performance.

Feasibility. It is the feasibility of a remote device technology for today's and the future real world. The difference of technical backgrounds is worthy of consideration, that is, the time between when the technology was proposed and the time when the technology is used. Most remote device technologies were developed many years ago and have been used for a long time. One of the reasons is that these technologies have been feasible enough to be implemented and deployed for contemporary hardware and software. The deployment feasibility of remote device technologies should be reviewed from the viewpoints of hardware cost, software complexity, modularity, maintenance cost, and extensibility.

2.3 Related Work

Many technologies in distributed systems work for accessing remote devices. In this section, these technologies are categorized into four remote device architecture categories. These categories are distinguished by their device abstraction approaches, which are based on hardware and software of their targets.

2.3.1 Peripheral I/O Bus

A peripheral I/O bus provides remote device access to a degree.

USB, IEEE1394, and Bluetooth

In the last half of 1990s, multipurpose peripheral buses were developed to connect computers and various devices, which were based on the progress of hardware implementation technologies. USB (Universal Serial Bus) [24] is a peripheral interface for most types of peripheral devices of a computer. A series of 125 us I/O transactions are scheduled for data transfers to (from) devices, which includes bulk data for storage devices (i.e., asynchronous transfer), sporadic data for interruptive devices, and isochronous data for multimedia devices (i.e., synchronous transfer). IEEE1394 [1] is a serial I/O bus which connects up to 63 nodes of computers and devices in a tree or a daisy-chained topology. It supports synchronous and asynchronous data transfer at 400 Mbps or 3.2 Gbps (IEEE1394b [2]). Each transfer is scheduled by a data transfer cycle of 125 us. Bluetooth [14] is a radio

communication technology that connects computers and devices by FHSS (Frequency Hopping Spread Spectrum) in the 2.4GHz ISM (Industrial Scientific and Medial) Band. A group of nodes synchronized to a frequency hopping sequence is called a piconet, in which one master node communicates with up to seven slave nodes. It supports synchronous and asynchronous data transfer at 1-2 Mbps within a range of 100 meters.

USB, IEEE1394, and Bluetooth are considered to remote device technologies that can provide computers with remote access to devices on a bus. These technologies allow computers to control devices directly without any abstraction for remote access. I/O messages between a computer and a remote device are independent of operating system structure, and any computer with a peripheral interface of this kind can access a remote device on a bus.

However, basically, a device is shared in an exclusive way except that some IEEE1394 storage chips support concurrent access for shared storage applications such as cluster file systems (e.g., OCFS [30], GFS [89], and Lustre [22]). Minimum disconnected operations are supported by bus hardware and its device driver; an operating system is just notified of a node joining and leaving, and applications need to handle them by themselves. Remote device access is limited under physical link topologies.

SCSI

SCSI (Small Computer System Interface) [95] is a specification of hardware interfaces and an I/O protocol family mainly used for connecting storage devices to computers. It was intended to connect not only storage devices but also other types of devices such as printers and scanners. Today, the market for the SCSI interface is focused on storage access of high-performance computers because of the advent of other low-cost interfaces. However, the SCSI command is widely employed for storage access over various physical media, because the SCSI architecture model makes logical I/O commands independent of transport media (e.g., the SCSI parallel interface, Fiber Channel, and IP network [85]).

The SCSI technologies are considered as a remote device technology that provides remote access to storage devices over Fiber Channel network or IP network. A storage device is abstracted by SCSI commands that are interoperable among

operating systems. Multiple computers (*initiators* in SCSI) share one storage device (*target*) concurrently, if supported. An intelligent disconnected operation is not covered in this level.

IPMI

IPMI (Intelligent Platform Management Interface) [53] is an integrated platform for monitoring and management of computer hardware. Hardware status, such as mainboard temperatures and fan speeds, is monitored via an IPMB (Intelligent Platform Management Bus) and aggregated into a BMC (Baseboard Management Controller). An IPMB can be also connected externally to an ICMB (Intelligent Chassis Management Bus) to correct other external computers' statuses into the BMC. The power supply of this system is independent of target computers in order to monitor them all the time. A BMC is accessed from management programs in local or remote computers. For example, SNMP [16] is applicable to access remote hardware status corrected by IPMI.

IPMI is focused on simple and low-speed devices such as sensors and BIOS consoles. These devices can be easily abstracted by simple I/O messages and transferred over hardware bus and IP networks. Its hierarchical design absorbs the difference in physical topologies, and the specifications standardize its message and interface to be interoperable between various vendors' hardware.

I2O

I_2O (Intelligent I/O) [88] is a peripheral interface specified by I2O SIG in 1996. I/O processors are deployed on peripheral interface hardware, which process interrupt handling and I/O transfers to improve I/O performance by offloading. Device drivers of I_2O are split into two parts, HDM (Hardware Device Module) and OSM (Operating System Module), by specifying a common API and device class (e.g., random block storage, tape storage, SCSI peripheral, and LAN). In OSM, device class drivers are developed by operating system vendors, which are independent of vendor's implementation details of a class device. In HDM, lower level drivers are developed by device vendors, which are close to hardware and independent of class driver implementations by operating system vendors. This

split driver model aimed to decrease driver development costs for both device vendors and operating system vendors.

A host-independent communication model is designed between OSM and HDM, and I/O messages are transferred by a transport mechanism. A PCI bus is the only transport actually used for it. However, I2O SIG had a plan to define transport layer support for ATM and Fiber Channel [78]. Although I_2O did not diffuse widely and its development seems to have stopped, interoperable device access over networks could be supported by its architecture.

2.3.2 Network-Attached-Peripherals Computer

Multimedia computers composed of network-attached peripherals had been studied mainly until the middle of 1990s. They aimed to process real-time and large multimedia data efficiently. Peripheral devices are physically connected to a high-speed network with end-to-end QoS guarantee, and I/O data is transferred directly between devices without going through a processor node. ViewStation [96], and Desk Area Network [42] utilize ATM. Early on, Netstation [32] employed the ATOMIC high-speed switched LAN, which is the base of Myrinet [74], and later switched to IP network [47, 69]. This architecture is also discussed for a mobile multimedia computer with efficient energy management [41].

In these technologies, remote device access over a network is provided to processor nodes. Compared with other remote device architecture, this approach is distinguished by the fact that a computer itself is composed of available peripheral devices in the network and that devices are fully decoupled from peripheral interfaces. Ideally, this approach makes it possible by further studies that an ad-hoc computer is organized on demand adapting to situations of users and networks. However, in recent years, processing power and peripheral bus speed have progressed so much that this computer architecture, because of an I/O bottleneck of multimedia data, did not spread in the consumer market.

2.3.3 Operating System

In operating system research, remote device access has been studied for many years. Since an operating system abstracts computer hardware to resources, all

remote access to resources implies remote device access. For further discussion, representative remote device technologies on operating systems are reviewed to present how device hardware is abstracted to be accessible over networks.

LOCUS

LOCUS [99] is a distributed operating system proposed by University of California at Los Angels in 1983. It supports transparent data access through a network-wide file system, automatic storage replication, transparent distributed process execution, and UNIX compatibility. It allows transparent use of remote devices in most cases, except for raw non-character (i.e., unbuffered block) devices accessed by executing process remotely. Since LOCUS was developed before the abstraction layer of file operations like Vnode [60] was introduced into UNIX, remote access was implemented by each device driver.

V

The V kernel and the V distributed system [17] were developed by Stanford University in the early 1980s. The V kernel is a micro-kernel operating system that provides transparent IPC between distributed processes (known as threads in later micro-kernel architecture) on machines in a local area network. The device server is a pseudo-process that enables device access for other processes through IPC. Since the V inter-kernel protocol is a transparent extension of the local V kernel IPC, all remote device access is transparently possible by communicating with a remote device server through the IPC. It supports Ethernet interfaces, serial lines, mice, disks, and consoles through an object-based protocol (i.e., the V I/O protocol). The V I/O protocol abstracts device I/O not as stream I/O but as block I/O. A current file position is not maintained in a device server by specifying target blocks in read and write operations in order to allow the operations to be idempotent. As discussed in its paper, the use of the I/O protocol means that the kernel specifies a service-level protocol rather than just defining transport level communication. The base layer of device abstraction is restricted by the I/O protocol. The device abstraction of remote device access is also affected by this restriction.

Sprite

Sprite [75] is an experimental network operating system designed for networked uniprocessors and multiprocessors with large physical memories. It was developed by the University of California at Berkeley from 1984 to the early 1990s. It provides a transparent network file system, shared memory between processes on single workstations, process migration between workstations, and the 4.3BSD UNIX compatibility.

Separating I/O operations and naming in Sprite provides remote access to all the resources easily such as peripheral devices and IPC channels as well as regular files [100]. Sprite allows users transparent access to remote I/O devices via special files on its file system, viewed as a single system image, regardless of a device's location. This mechanism is implemented by a kernel-to-kernel RPC facility and a name space management technique called a prefix table. Remote device access is performed by mapping the pathname of a remote device to attributes of a remote object of the device, and then creating an open I/O stream to the object from the attributes.

CHORUS

CHORUS [82] is a distributed operating system developed by INRIA in France from 1979 to 1986 and by Chorus systems from 1987. CHORUS consists of a small distributed real-time Nucleus and a set of subsystem servers. Nucleus is a micro-kernel that supports location transparent message-passing, thread scheduling, and virtual memory operations. A UNIX System V interface called CHORUS/MiX was developed as a subsystem of CHORUS. Unlike other contemporary UNIX subsystems in micro-kernel operating systems, UNIX drivers are encapsulated into servers on Nucleus by the emulation of kernel functions for transparently extending traditional UNIX services to be distributed [5]. For example, a remote disk is accessed by sending an IPC message from the local server of a file system driver to the remote server of its disk driver. This feature allows remote driver access, co-existing subsystems, and dynamic loading and unloading of drivers, for a fault tolerant storage system [59].

Mach

Mach [4] is a micro-kernel operating system developed by Carnegie Mellon University from 1985. It aimed to be a scalable distributed system of heterogeneous hardware, such as one to thousands of processors and various memory systems. It supports integrated virtual memory management and interprocess communication, and multiple threads of control within one address space.

The I/O subsystem of Mach 3.0 [33] released in 1990 was redesigned to support device-independent and location-independent drivers implemented in userland. This approach reduces significantly the hardware-dependent code of the Mach kernel and permits users to manage remote devices transparently. Since different operating system environments on the Mach kernel have their own abstractions developed not by application programmers but by system programmers, hardware-dependent drivers in the kernel provide the lowest level abstractions possible. Device-independent drivers are classified into groups such as screen, console, disk, tape, serial line, and Ethernet. Portability and code sharing are greatly increased by this structure.

Solaris MC

Solaris MC [57] is a distributed system for a cluster of computers connected by a high-speed interconnect, developed by Sun Microsystems in the mid-1990s. It provides a single system image that appears to users and applications as a single computer running the Solaris operating system. Its components are implemented in C++ through a CORBA-compliant object-oriented system. It supports a location-transparent distributed file system with a caching mechanism and remote process execution. These mechanisms are based on object-oriented implementations and distributed techniques by the Spring OS [72], which was formerly studied by Sun Microsystems in the early 1990s.

Remote device access among nodes is supported by its I/O subsystem, which enables any I/O device to be accessed from any node without regard to the physical attachment of devices to nodes. Applications on a node are able to access devices as local devices even when the devices are physically attached to another node. Drivers are dynamically loaded per node to which a device is physically attached. Process context is assigned to the drivers accessed from remote nodes;

this provides a logical equivalence between local and remote access in order to utilize existing drivers without modification. Uniform device naming is achieved on its file system, which replaces Vnode architecture through an object-oriented interface [67].

Other operating systems and techniques

As described above, most operating systems for a single system image have aggressive support of remote device access through their cluster-wide file systems for remote process execution and process migration. Remote device support tends to be noted in the explanation of these systems. VAX cluster [62] and OpenSSI Linux [97] are other examples of this sort of system.

Micro-kernel operating systems with transparent IPC over network also tend to have remote device support, by passing I/O messages to remote server threads of device drivers and also by separating naming from I/O mechanisms. QNX [44] and Inferno [98] are other examples, which are commercial products today for embedded systems.

A lot of distributed file systems have been widely studied for long years [65]. Remote storage devices are accessed through the abstraction of a file on these file systems. For example, NFS [92] and CIFS (Common Internet File System) are the most widely-used network file systems. AFS [48] by Carnegie Mellon University is known as a scalable distributed file system on local and wide area networks. Coda [86], also developed by Carnegie Mellon since 1987, is a well-known distributed file system focused on mobile computing. It utilizes not only AFS techniques but also more advanced support for caching and disconnected operation.

On the other hand, some file systems support not only regular files but also special files such as device files and message pipes, consequently allowing remote access to devices other than storage devices. Some of them were already described above. RFS [81], included in AT&T System V Release 3.0 (SVR3) in 1987, is another network file system that supports remote device access via device files on the file system. It exploits the File System Switching and remote system call mechanism in SVR3. Plan 9 [76], developed by Bell Laboratories since the mid-1980s, strongly abstracts any computer resources as files. Compared with the

UNIX systems, resources including devices are highly abstracted as files to be controlled easily from users and applications, so that its distributed file system allows remote device access by passing simple control and data messages to device files. These messages are transferred by the light-weight network communication protocol 9P. This design decision can be performed because Plan 9 is intended to explore a new operating system model rather than to extend existing systems with compatibility.

2.3.4 Middleware and Application

Middleware is a software platform that often exists between applications and underlying platforms such as operating systems and hardware. It provides common services and interfaces for applications to improve software productivity overall. Although the term middleware is used in various ranges of context, a definition is given in [8] as follows:

> *A middleware service is a general-purpose service that sits between platforms and applications.* (...) *Middleware components have several properties; they are generic across applications and industries, they run on multiple platforms, they are distributed, and they support standard interfaces and protocols.*

Conventional middleware systems are customarily classified into four categories: remote procedure calls such as DCE RPC (Distributed Computing Environment RPC) and Sun RPC [90], object-based middleware (e.g., CORBA [38], DCOM (Distributed Component Object Model), and Java RMI (Remote Method Invocation)), message-based middleware (i.e., data exchange systems by message queues), and transaction processing monitors (i.e., database systems). Also, recent middleware research addresses adaptability and reflection [83, 84]. These techniques are applied to mobile computing [36], sensor networks [79], and networked home appliances (e.g., UPnP [34], HAVi [40], Jini [70], and ECHONET [28]).

These technologies can offer remote access to device-derived resources. Remote devices are utilized through elaborated interfaces designed for features of a

middleware system. As long as applications are built on the middleware system, remote device access is possible in its manner of device resource abstraction.

In a broad sense, any network application is considered a remote device technology if it offers resource access over networks. For instance, VNC [80], RDP (Remote Desktop Protocol), and X Windows System [87] are remote desktop applications. LPD [50] and Internet Printing Protocol [63] are remote printing technologies. Telnet [68] and SSH [102] are regarded as remote console technologies. HTTP [31] and FTP [77] are used for remote storage access.

2.4 Summary

This chapter presented diverse ranges of remote device access. It covered hardware-level approaches, system software-level approaches, and application-level approaches. In remote device technologies, a service interface of remote device access is based on underlying hardware and software architecture.

Peripheral I/O buses provide *remote* device access. Coverage is often limited by specifications of their physical media. However, SCSI and IPMI are designed to be independent of transport media, so that they have the option of remote access over an IP network. Since specifications define I/O interfaces and messages well, peripheral I/O buses are interoperable among different computers and operating systems. USB, IEEE1394, and Bluetooth are general-purpose serial I/O buses with both synchronous and asynchronous transfers. They support not only storage devices but also multimedia devices that require real-time data transfers. Also, they are now widely available in the consumer electronics market. Compared with software-based approaches to remote device access, the I/O bus mechanisms seldom offer certain features such as concurrent access, aggressive disconnect operations, and caching mechanisms, by themselves.

Multimedia computer systems were studied for efficient processing of large real-time data. Peripheral devices are directly attached to networks where processing units are available. This architecture involves the possibility of flexible remote device access in distributed systems. However, there is a big architectural gap from today's computer architecture, which prevents achieving this computer architecture in the near future. The technical issues that motivated these studies

are now alleviated by the progress of hardware.

A lot of operating systems address remote device access by using their IPC or RPC mechanisms to driver threads or driver procedures. Microkernel operating systems can be extended flexibly by putting parts of their system components in other processor nodes. They have the possibility to make device access reconfigurable and adaptable in distributed systems. However, basically, the methods of device abstractions are deeply dependent on the structures of operating systems. Even if remote device access for various devices is allowed by an operating system, its I/O protocol is not interoperable between different operating systems, except for some standardized protocols of storage. Although diverse operating systems on various computers interact with an increasing number of devices, existing remote device access technologies on operating systems lack either device generality or interoperability.

Middleware systems are widely used to provide uniform access and advanced features for distributed resources. They can offer integrated services for remote device access, such as I/O message transfers, naming, security, and discovery. However, remote device access is given to only applications which exploit middleware systems. A lot of existing operating systems and applications not built on them are out of the scope of their support. Network applications are regarded as remote access to device-derived resources. However, these applications are not general-purpose software that can support devices and usages other than their targets.

Lacking is a flexible device access technology that enables remote access to any devices smoothly for any computers and applications in an interoperable way without any modification. Consequently, it is necessary to explore alternative remote device access technology, which can provide reconfigurability and adaptability of device usages for distributed systems.

Chapter 3

USB/IP

In operating systems, the I/O interfaces are based on their own semantics of system design. If a remote device technology just extends the I/O interface of an operating system straightforwardly, it lacks interoperability among different operating systems. Mach 3.0, which allows transparent IPC, has an I/O interface that provides seven remote procedure calls for device drivers. However, this semantics is not applicable to other operating systems as it is, that is, fully specific to its driver designs. The V kernel defines its I/O interface to be applicable for remote access (i.e., block-based access and idempotent read operation). However, this semantics also deeply depends on its system design, which is difficult to apply to other operating systems for remote device access. For the same reason, RFS, which fully extends the UNIX semantics, never becomes an interoperable solution if implemented in other operating systems. Not only `ioctl()`-like operations [1] but also the designs of I/O interfaces essentially include parts specific to each operating system.

In some remote device technologies, all operating systems give up keeping their own semantics for remote device access (e.g., NFS). However, this approach results in lacking the same semantics of remote devices as physically-attached ones. Full functionality cannot be provided for remote device access. In particular, this deficiency is noticeable for the types of devices handled in different ways inside operating systems; operating systems cannot bring out all functionality of

[1] For instance, `device_set_status()/get_status()` in Mach 3.0, `UioControl()` in the V kernel, and `IoControl()` in Sprite.

devices in their own manner.

iSCSI is based on the transport-independent design of the SCSI architecture model. However, a multipurpose peripheral bus that uses IP networks for its transport is still lacking. Further studies are needed to discuss such peripheral bus models and operating system support.

Through discussions about remote device technologies in operating systems, the key to designs is considered to be the location (layer) of a network extension mechanism in the peripheral I/O paths of operating systems. This thesis proposes USB/IP, a network extension of the USB model. It is focused on its host-controller-based peripheral interface and aims to extend physical bus topology virtually by utilizing a virtual bus driver. Although some parts of its mechanism are similar to straightforward remote procedure calls to remote device drivers, it can avoid the potential complexity that sometimes exists in remote device techniques when supporting diverse types of devices with interoperability among different operating systems. In the proposed mechanism, the request types used in remote device access are basically reduced to just two primitive ones; pushing out an I/O message to a device and getting it back. The I/O messages and the method of messaging become essentially independent of the operating systems. The semantics of remote requests basically results in the same one as local requests to physically-attached devices. [2]

First, the design of the USB model is explained, and then its potential difficulty in network extension is discussed. Next, its prototype design is presented, which addresses inevitable semantics differences by preserving the USB model as much as possible. Its performance evaluation is also described to show its feasibility and performance models.

3.1 USB Design Model

USB (Universal Serial Bus) is one of the more sophisticated peripheral interfaces, providing serialized I/O, dynamic device configuration and universal connectiv-

[2] In a strict sense, small semantic modifications for remote devices will be involved in the design details. These are the keys for giving insight into the peripheral models and operating systems for the network extension.

ity. The USB 2.0 specification, announced in April 2000, specifies that a host computer can control various devices using three transfer speeds (1.5 Mbps, 12 Mbps, and 480 Mbps) and four 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 sending sound data to a USB speaker. The shortest I/O transaction interval supported using Isochronous transfers, called a microframe, is 125 us. 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 480 Mbps mode, 20 % of the total bandwidth is reserved for Control transfer. The Bulk 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[3] of USB is layered as illustrated in Figure 3.1. 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[4] (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

[3] In this thesis, some technical terms in the USB model are changed from the USB specifications for readers.

[4] Client Software in the USB specification.

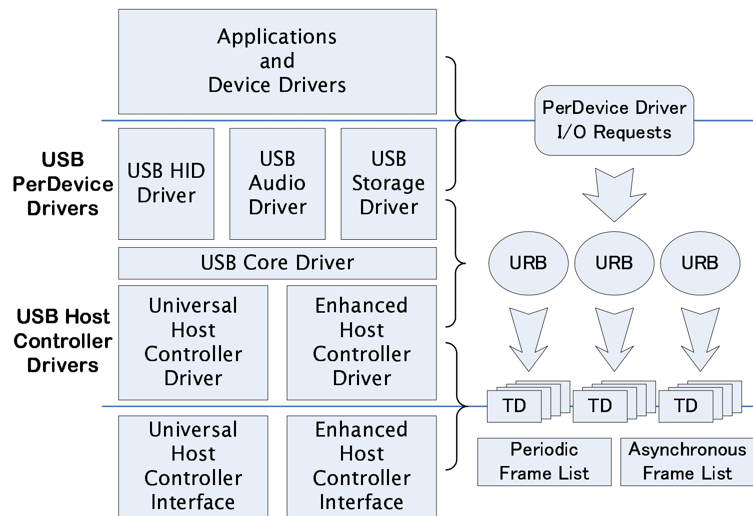

Figure 3.1: USB Design Model

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[5] 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.

A typical implementation supports request queueing to improve I/O performance. A URB is allocated with its completion handler, and completion of a request is notified by calling its handler. This is an asynchronous process from the viewpoint of the I/O model.

[5]USB Driver (USBD) in the USB specification.

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.

3.1.4 USB Request Block

In the USB model, a USB Request Block[6] (URB) presents USB device I/O in a controller-independent form. It basically includes the following members:

- target device address
- target endpoint
- I/O buffer.

A URB is defined in an operating-system-dependant manner. However, the following members are normally associated with or included in it:

- direction (Input/Output)
- speed (1.5 Mbps/12 Mbps/480 Mbps)
- transfer type (Control/Bulk/Interrupt/Isochronous)
- payload size
- additional information (e.g., bandwidth management and miscellaneous flags)
- completion handler.

[6] I/O Request Packet (IRP) in the USB specification

A driver stack works as the following cycle:

1. A USB PDD converts I/O requests from another driver into URBs and submits them to a USB HCD via the USB Core Driver.

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.

Most USB driver stacks allow a USB PDD to enqueue multiple URBs simultaneously for I/O performance improvement.

3.2 Requirement

The proposed mechanism involves several design criteria and considerations to maximize its benefits for remote device access. USB was not designed considering remote device access. The below criteria need to be tackled in its system design. This research also shows the limitations of the USB model for the network extension.

Full Functionality. All functions supported by remote devices should be manipulated by operating systems. The extension mechanism should conceal only bus differences and should not affect per-device operations. In kernel structure, both locally-attached devices and remote devices should be positioned in the same layer.

Network Transparency. This dissertation defines network transparency as: "remote devices can be controlled by existing operating systems and applications without any modification." The extension mechanism should conceal its implementation details, and remote devices should be controlled by existing device drivers and applications without modification. Ideally, other components of operating systems and applications should not notice any difference between remote devices and locally-attached ones.

Interoperability. The extension mechanism should be designed to be interoperable among different operating systems; it is based on the use of low-level

I/O messages and messaging in the USB model, which are the same as those of locally attached devices. These protocols, which are defined by several standards, are independent of operating system implementations, so that computers with different operating systems have the possibility that they can share and control devices remotely.

Generality. The extension mechanism should be designed to support all types of transfers in the USB mode, so that all types of devices can be accessed over IP networks in one extension mechanism.

Moreover, there are some issues that need to be considered when applying the extension mechanism in practice:

Concurrent Access. In the proposed mechanism, a remote device is exclusively accessed from computers; its mechanism does not provide special support for concurrent device access from clients. In contrast, most remote device technologies are designed to support concurrent access to device-derived resources. The proposed technology is implemented as an extension mechanism in the lowest layer of an operating system; both types of remote device technologies can be complementary to each other. The integration of both mechanisms is discussed in the next chapter by developing a device sharing application based on USB/IP.

Performance. The I/O protocol of USB is not designed for transmission over IP networks. Using these protocols over an IP network raises performance degradation against network delay and jitters, which are much larger than a wire cable of a peripheral bus. However, high-speed network technologies, such as Gigabit Ethernet, are now able to provide bandwidth that is of the same order as that of modern peripheral interfaces. In addition, the rapid progress of networking technologies will alleviate many of these issues in the near future.

Semantics Gap. The extension mechanism also involves the problem of I/O semantics: a peripheral interface, a peripheral device, and their device drivers are not originally intended for network extension. If these designs have difficulty being extended for remote access in a straightforward manner, the network extension needs to abstract physical device access and bridge semantics differences between local and remote access. However, these efforts tend to hide the physical interface of a remote device from operating systems, so that they cannot access remote devices in a fully transparent manner as if physically-attached.

3.2.1 Semantics Alteration

As in Section 3, the request semantics of USB/IP can basically be designed as a straight extension of I/O messaging to physically-attached devices via host controllers, thereby preventing the remote device mechanism from being contaminated by operating-system-dependent semantics. This means the remote device mechanism essentially becomes interoperable among operating systems. However, the design needs to be performed carefully for the criteria, since USB, designed only for its media, is never intended to be extended over IP networks. Actually, part of its remote request semantics is modified from the original USB model inside operating systems.

Time Constraint

Some requests have time constraints on their completion; a request should be completed by a host controller within a defined period. If a submitted request is not completed within timeout, a procedure call in drivers will return an error status. There is the possibility that timeout values in drivers and device firmware need to be increased for network extension. This problem applies to all device drivers, not only USB device drivers, and applications that interact with remote USB devices.

Scheduling

The USB model defines the minimum request scheduling policy in a host controller, in which periodical requests come in first before asynchronous ones with minor exceptions.[7] In addition, actual implementations of host controllers and driver stacks perform more advanced scheduling policies to maximize the efficiency of bus utilization and realtime processing [49]. However, such a request scheduling does not apply to the network extension; a strict scheduling policy designed for a physical host controller is difficult to achieve over networks, and another mechanism will be required for efficient bandwidth usages of networks and remote host controllers.

[7] A host controller reserves bandwidth for Control transfers: 10 % for Low/Full speed mode and 20 % for High speed mode.

Bandwidth

Periodical USB devices claim transfer bandwidth to a host controller; their embedded descriptors show several options of transfer modes (*configuration* and *interface*), one of which is selected by a device driver against available bandwidth. For USB/IP, the available bandwidth for a remote device is a least common multiple of available bandwidth of a remote host controller and networks on its path.

Synchronization

The USB model has a synchronization mechanism of I/O sources (`SYNCH_FRAME`), however, which is rarely implemented or exploited in driver stacks. Since this feature, not important for most USB devices, is more difficult to implement over networks, its support is dropped in the remote semantics.

Request Cancellation

The USB model allows a device driver to cancel or abort a pending request; this happens when, for instance, an application stops the usage of a device, a device is disconnected, or a device driver is unloaded. In addition, when a device returns an error code for a request to an endpoint, the request is aborted or retired, other pending requests to the same endpoint are retired, and no further request is accepted until its driver recovers the device and clears the halt state.[8]

Since the design details of interfaces and semantics about request handling are implementation-dependent parts of the USB model, a driver stack implementation may involve problems with the cancellation of a remote request, which are caused by a timing gap between remote and local stacks. First, if an error occurs in a remote device side, its driver may still submit new requests to a halted endpoint until the error is notified to a virtual host controller. Second, the cancellation of a request cannot be directly notified to a remote host controller driver; although a request pending in a client side can be cancelled synchronously, a request already sent to the remote host cannot be cancelled synchronously from a device driver. These issues will be noticeable in the case that the request semantics in a driver stack are not appropriate to be extended over networks.

[8] Section 5.3.2 Pipes in [24]

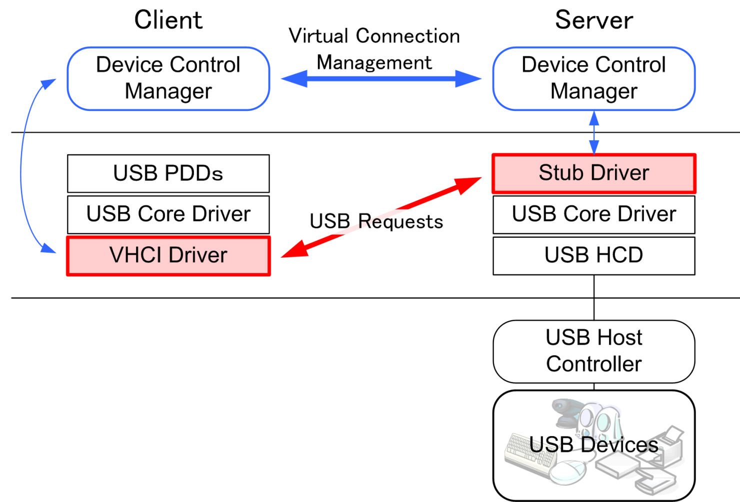

Figure 3.2: USB/IP Design

3.3 Design

The design of USB/IP is illustrated in Figure 3.2.

3.3.1 Virtual Host Controller Interface Driver

To implement the network extension of USB, a *Virtual Host Controller Interface* (VHCI) driver is developed as a virtual bus driver. 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 that has target USB devices. In the remote machine, a software mechanism decodes incoming USB/IP requests, extracts the URBs, and then submits them to physically-attached USB devices.

Any interface differences between directly-attached USB devices and remote USB devices are 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.

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.

3.3.2 Protocol Data Unit Based on USB Request Block

An IP network typically has a significantly larger transfer delay and more jitter than a 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 3 KB of data in every microframe (125 us), and the Bulk transfer type needs to transmit 6.5 KB of data in a microframe. Therefore, to transfer USB commands over an 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 until 10 ms, while still maintaining an I/O granularity of 125 us. Similarly, using the Bulk transport type, a URB that has a 100 KB buffer can be used to transfer 200 I/O transactions containing 512 B of data each.

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 480 Mbps 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. The evaluation of the use of URBs for USB/IP is written in more detail in Section 3.4.

3.3.3 Stub Driver

In this design, which aims to fit existing driver stacks, a *Stub* driver is added as a new type of USB PDD in the server side, which redirects USB requests to physically-attached devices. Since a Stub driver is implemented as a normal USB device driver, not as a special HCD driver or a modified USB core driver, its implementation becomes greatly simplified and independent of host

controller hardware. However, the USB requests that also change the status maintained in the driver stack need to be intercepted, so that data maintained by the USB Core Driver can be updated correctly, such as `SET_ADDRESS`, `CLEAR_HALT`, `SET_INTERFACE`, and `SET_CONFIGURATION`.

To implement the server side by a Stub driver, a USB driver stack needs to support non-blocking request queueing and a way of updating status consistently in both a device and drivers. In addition, it is appropriate that the stack allows a PDD to be bound to "a" USB device, not only to a logical interface in a device.

3.3.4 Prototype Implementation

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, ..)` of the USB Core Driver. After small sanity checks, `usb_submit_urb()` calls the `urb_enqueue(struct *urb, ..)` of the VHCI driver. The `urb_enqueue()` translates a URB into a `USBIP_CMD_SUBMIT` PDU (Protocol Data Unit) (described in Appendix) and then transmits it to a remote Stub driver. Next, the Stub driver receives the `USBIP_CMD_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 `USBIP_RET_SUBMIT` PDU that 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. Also, a `USBIP_CMD_SUBMIT` PDU can be canceled by a `USBIP_CMD_UNLINK` PDU as described in the Appendix. Figure 3.3 illustrates the components of the VHCI driver.

The semantics issues described in Section 3.2.1 are currently implemented as follows:

- **Time Constraint.** No timeout values in device drivers are increased. To minimize overhead of the network extension, URBs are transferred to enlarge I/O granularity over networks. In addition, the prototype is designed carefully without unnecessary memory copy.

- **Scheduling.** In Linux, a USB HCD needs to provide a request queue spe-

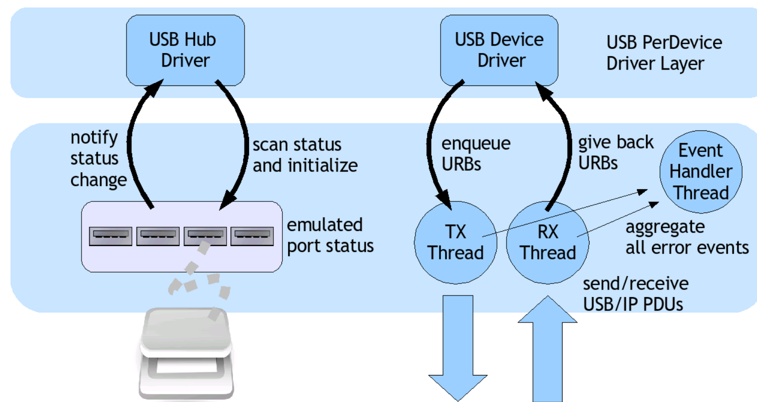

Figure 3.3: VHCI Driver Implementation

cific to each endpoint and process queues for periodical endpoints before asynchronous ones, except a special endpoint. Since a VHCI driver can aggregate remote devices attached to different hosts, the scheduling designed for a physical host controller is not applicable for it. Instead, it prototype implementation creates a request queue and its network transport for each remote device, even though remote devices are exported from the same host.

- **Bandwidth.** In Linux, available bandwidth of a host controller is checked when a USB driver starts I/O transfers. The overcommit of requests in a physical host controller is prevented by its USB driver stack in a server side; this results in an error code in a request. In addition, some USB PDDs also detect insufficiency of bandwidth by returned I/O messages and elapsed time. In the prototype, a VHCI driver emulates a host controller of High Speed (480 Mbps) support, and passes an error code about bandwidth from the server side to a USB PDD.

- **Synchronization.** The prototype does not implement this feature, although only a vendor-specific driver of an audio device exploits it to specify transaction timing in a millisecond order.

- **Request Cancellation.** The recent Linux USB driver stack provides an asynchronous interface to cancel a URB; its success or failure is notified

asynchronously to a caller of `usb_unlink_urb()`. The cancellation interface for USB HCDs is also an asynchronous call (i.e., `urb_dequeue()`). The prototype implements the cancellation of a request by exploiting these interfaces. However, the early version of the prototype developed for Linux kernel 2.4 could not implement this feature correctly, since a USB HCD in its USB stack also needs to implement a synchronous call to cancel a URB.

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 [73] [9] 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 data arrival and retransmit lost packets in order.

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.

As a practical application of USB/IP, a device sharing system for LAN environments has been developed. (Figure 3.4). For service discovery, Multicast DNS [19] and DNS Service Discovery [18] are used to manage the dynamic device name space in a local area network. An example usage of this application is illustrated in Figure 3.5. This section focuses primarily on the implementation of USB/IP. An application of USB/IP for device sharing is given in Chapter 4.

3.4 Evaluation

This section describes the characteristics of the USB/IP implementation. In particular, several experiments were carried out to measure the USB/IP performance.

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

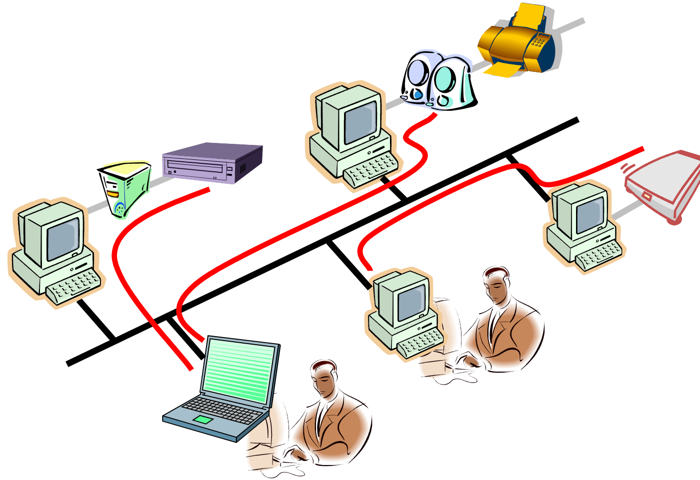

Figure 3.4: USB/IP Application
(Device Sharing for LAN)

```
% 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 3.5: USB/IP Application Usage

Table 3.1: Machine Specifications for Experiments

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

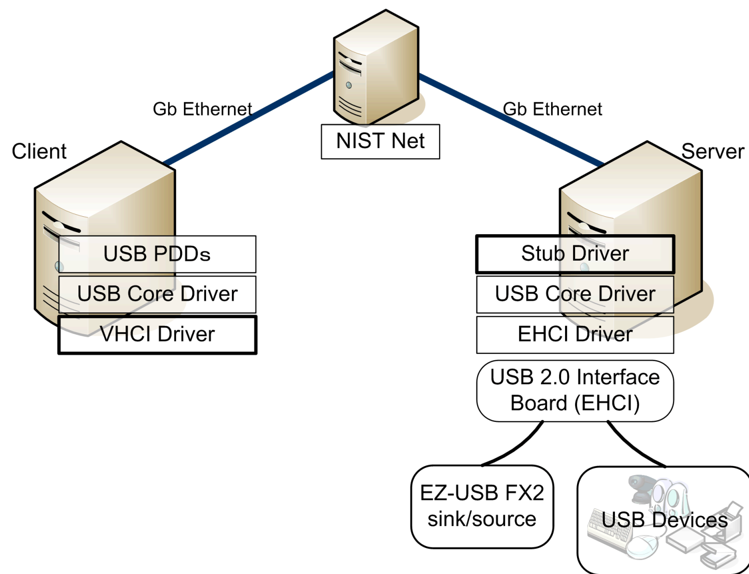

Figure 3.6: Experiment Environment

The computers used for the evaluation are listed in Table 3.1. To emulate various network conditions, the NIST Net [15] package on Linux Kernel 2.4.18 was used. A client machine and a server machine were connected via a NIST Net machine, as shown in Figure 3.6. Both the client and server machines run Linux Kernel 2.6.8 with the USB/IP kernel modules installed.

3.4.1 Performance Evaluation of USB Pure Sink/Source

In the first evaluation, a pure sink/source USB device is used rather than a real USB device in order to identify the performance characteristics of USB/IP itself.

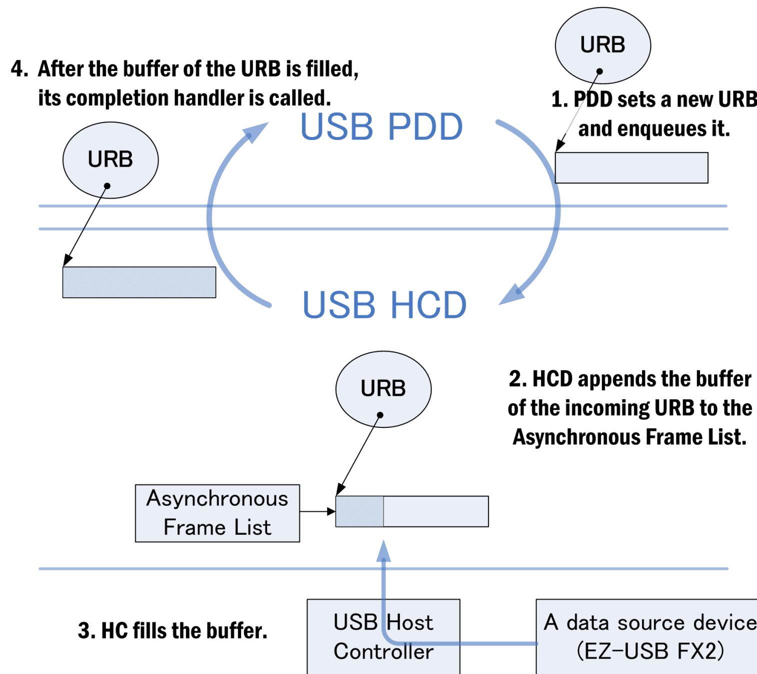

Figure 3.7: Driver Behavior of Asynchronous Transfers

A USB peripheral development board with a Cypress Semiconductor EZ-USB FX2 chip [25] 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.

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. The second evaluation 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 3.7 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 for processing a URB in the

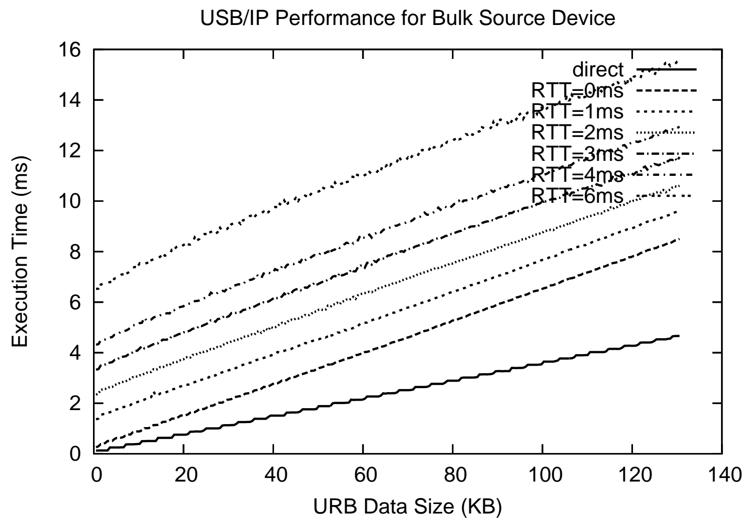

Figure 3.8: Execution Time of a URB

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 0 ms, the actual RTT between the client and server machines is 0.12 ms, as determined by `ping`.

The results are shown in Figure 3.8. From the graph, it can be seen that the relationship between the execution time of URBs and different data sizes is linear with a 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 the `TCP_NODELAY` socket option is used. 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 RTTs. The throughput *thpt* is then

$$thpt = \frac{s}{t_{overIP}} = \frac{s}{a_{overIP} \times s + RTT}. \tag{3.1}$$

A regression analysis shows a_{overIP} is 6.30e-2 ms/KB with a y-intercept for the 0 ms case of 0.24 ms. The throughput modeled by Equation (3.1) is shown in

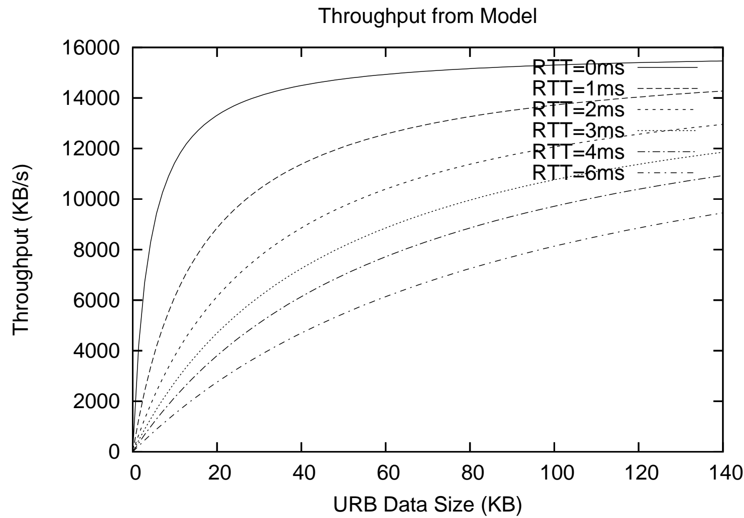

Figure 3.9: Throughput from Model

Figure 3.9. The actual throughput from the experiments is illustrated in Figure 3.10, 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 0.07 ms. This implies a relatively constant throughput of approximately 28.5 MB/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 (125 us). In 480 Mbps mode, one Bulk I/O transaction transfers 512 B of data to a USB device. In this case, the throughput is equal to $(7 \times 512B)/125us = 28MB/s$.

To summarize these experiments, the appropriate URB data size under various network delays has been estimated. The experiments 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,

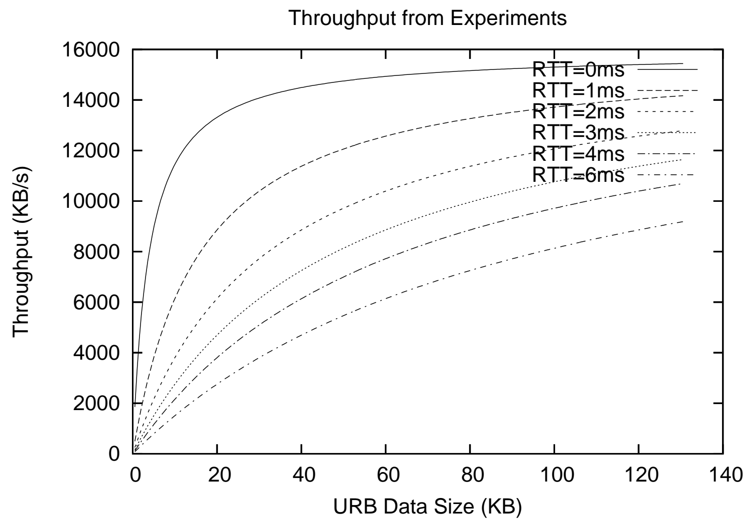

Figure 3.10: Throughput from Experiments

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.

Isochronous Transfer. This section examines 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, the firmware and a test device driver for an Isochronous source device were developed. The device and drivers are configured as follows:

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

In this case, the completion handler of the URB is called every 1 ms ($125us \times 8$). This 1 ms 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 10 ms, which is an acceptable trade-off between smoothness of

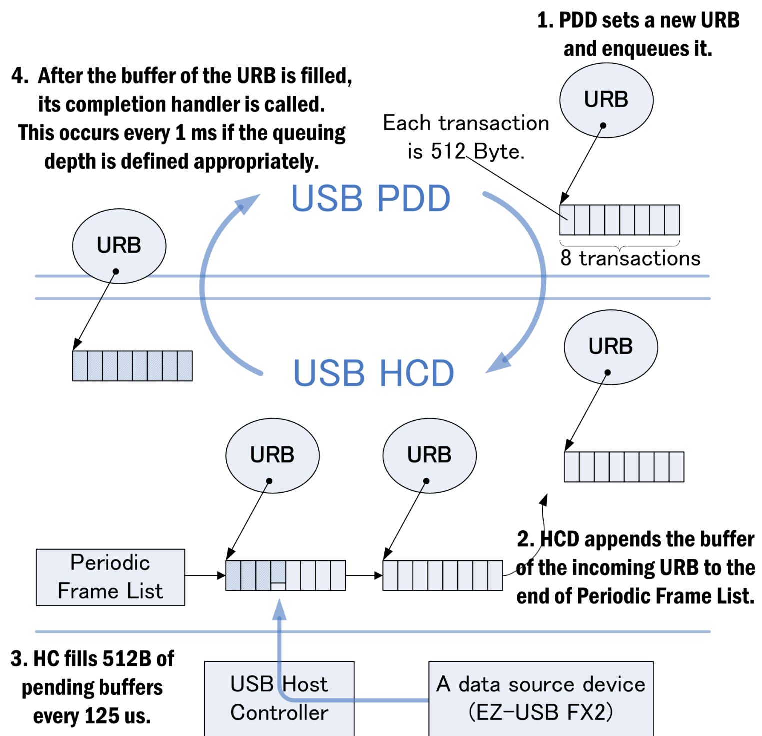

Figure 3.11: Driver Behavior of Periodical Transfers

I/O operations and the processing cost. Figure 3.11 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 eight 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 1 ms. 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 11.1 ms because of request starvation. When the USB PDD submitted two or more URBs simultaneously, the completion intervals were 1 ms, with a standard deviation of approximately 20 ns for any queuing depth.

Figure 3.12 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 1 ms completion intervals, provided that an adequate queuing depth is specified. For example, when the network delay is 8 ms, the appropriate queuing depth is 10 or more. Figure 3.13 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 1 ms. Figure 3.14 shows the standard deviations of the completion intervals. With sufficient URBs, the measured standard deviation is less than 10 us, including the NIST Net's deviations [15]. These values are less than one microframe interval (125 us) 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 1 ms. 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 3.15. 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

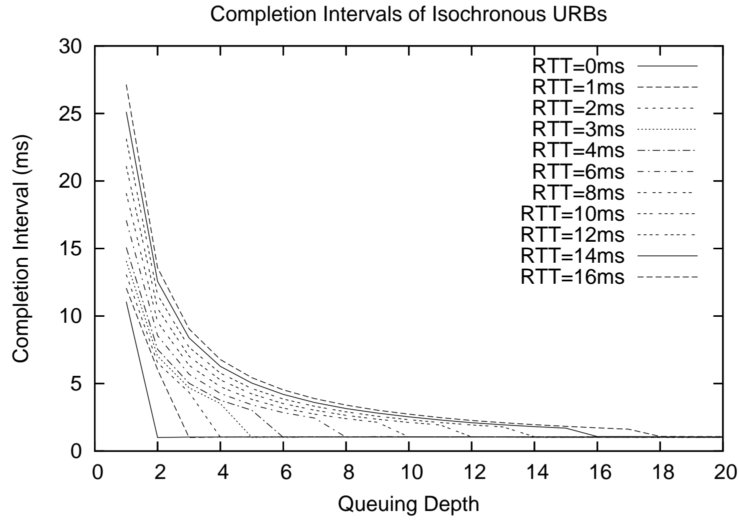

Figure 3.12: Mean Completion Intervals

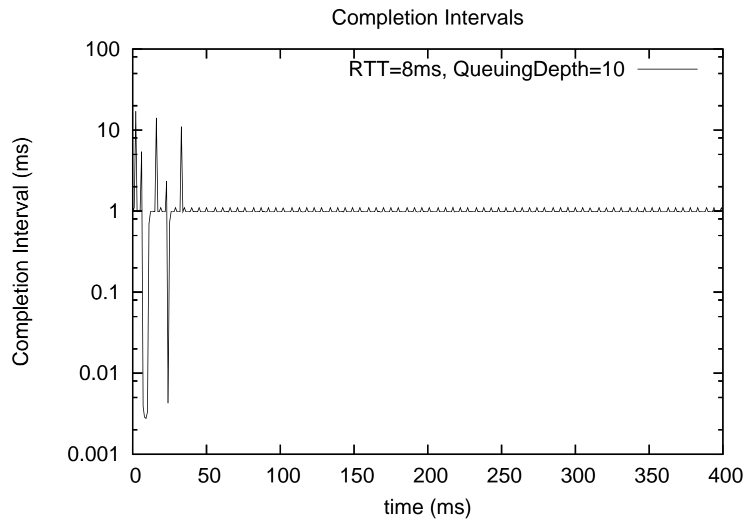

Figure 3.13: Time Series Data of Completion Intervals
RTT=8ms, Queuing Depth=10

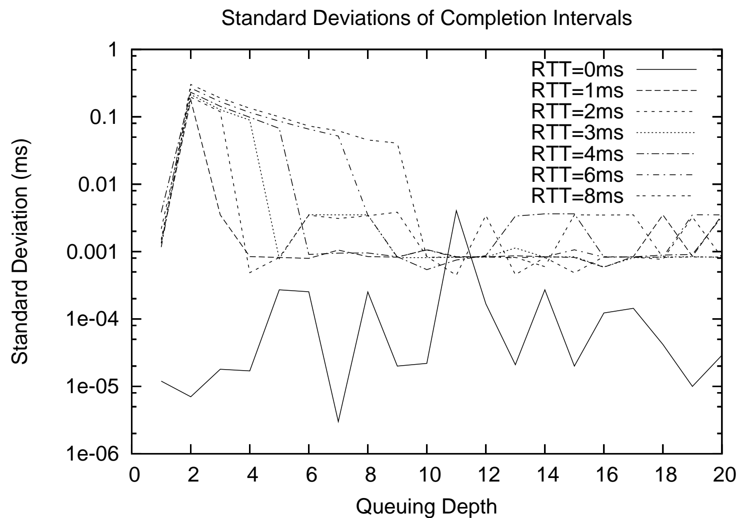

Figure 3.14: Standard Deviations of Completion Intervals

in which the next URB is pending is

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

The appropriate queuing depth q can then be calculated by

$$q > \frac{RTT}{x} + 1. \tag{3.2}$$

Comparing Figure 3.12 and Equation (3.2) 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 (3.2) 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 3.4.2.

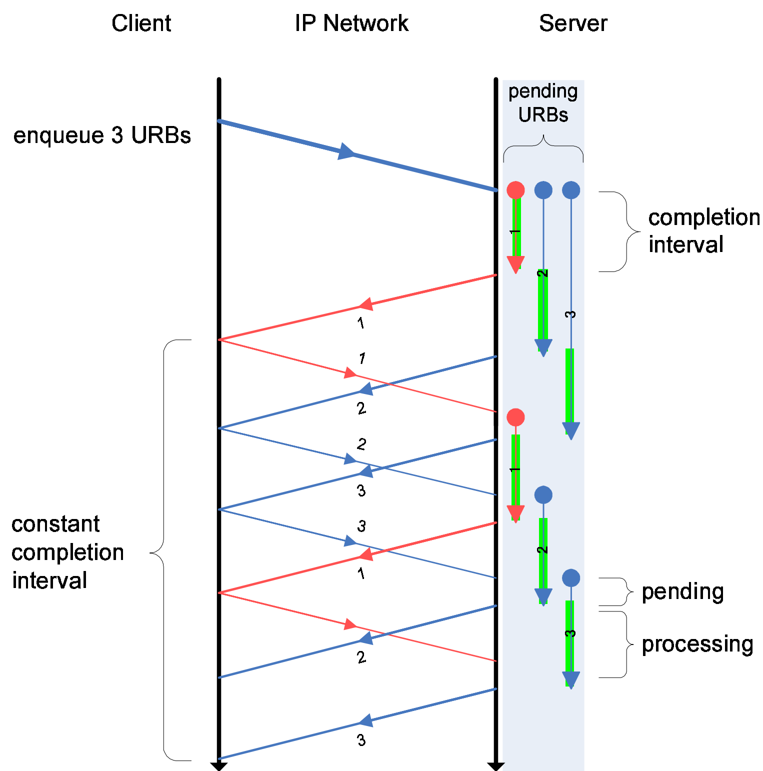

Figure 3.15: USB/IP Model for Periodical Transfers

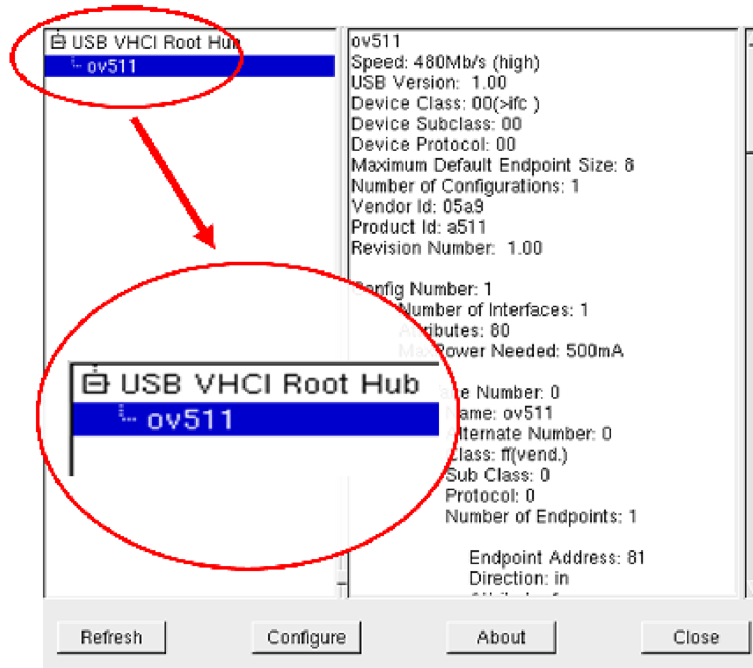

Figure 3.16: Output Example of USB Utility
`usbview` shows that a remote USB device is connected to a computer.

3.4.2 Performance Evaluation of USB Devices

This section examines the USB/IP characteristics for common USB devices. All USB devices tested can be used as USB/IP devices. Figure 3.16 shows a client host attached to a remote USB camera through the VHCI driver. In this case, the USB device viewer `usbview` [61] sees the device descriptors as if the camera were locally attached. The only difference 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 LAN environments, 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.

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

Table 3.2: Specifications of Tested USB Hard Disk

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

`mount/umount` operations, and perform normal file operations. Moreover, 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 3.4.1, the USB/IP performance of USB Bulk devices depends on the queuing strategy of Bulk URBs. The original USB storage driver of Linux Kernel 2.6.8 has been tested to show its effectiveness for USB/IP.

The experimental setup for this test was the same as that described in Section 3.4.1. NIST Net was used to emulate various network delays. The Bonnie++ 1.03 [23] benchmarks were used for an ext3 file system on a USB hard disk. The disk specifications are shown in Table 3.2. 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.

Figures 3.17 and 3.18 show the sequential I/O throughput and the corresponding CPU usage for USB and USB/IP, respectively. Figure 3.19 shows sequential 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 0 ms (0.12 ms 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 that 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

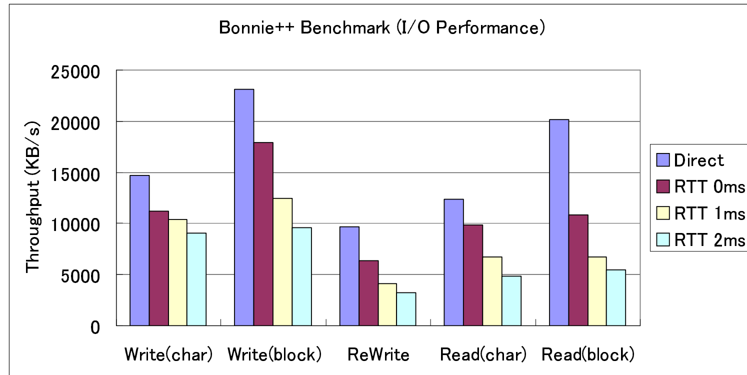

Figure 3.17: Sequential Read/Write Throughput

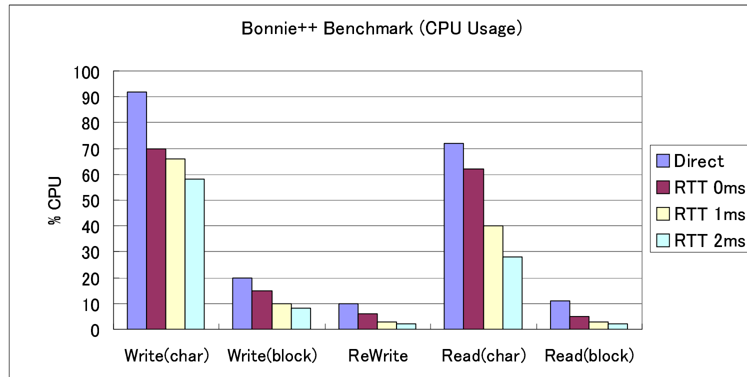

Figure 3.18: Sequential Read/Write CPU Usage

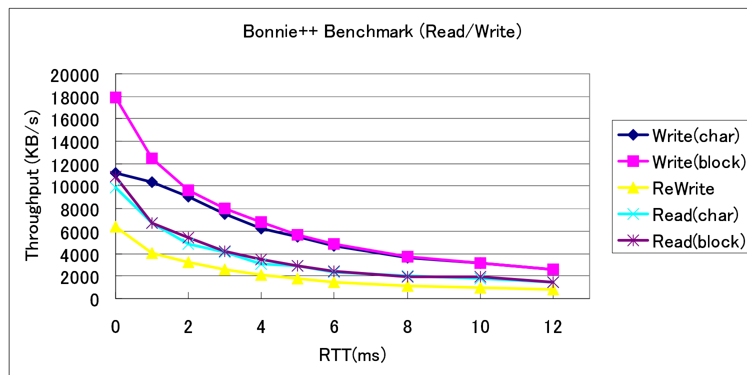

Figure 3.19: Sequential Read/Write Throughput (RTT)

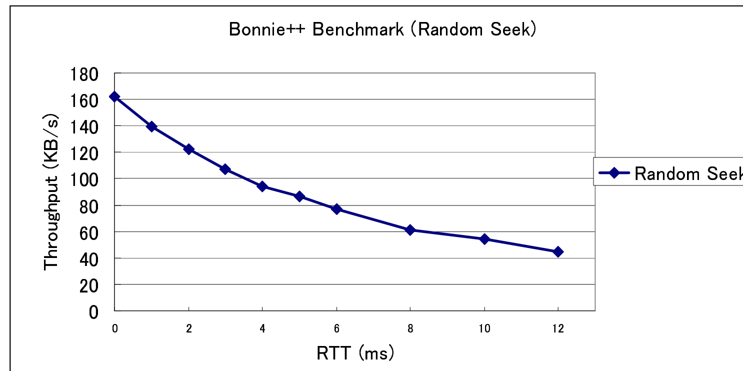

Figure 3.20: Random Seek Speed

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 128 KB. This queuing data size is small for USB/IP under some network delays, as discussed in Section 3.4.1. 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 3.20. 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 167 KB/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 tested is 5400 rpm and its seek speed is approximately 10 ms.

Figure 3.21 shows the speed of file creation and deletion operations by USB/IP under various network delays. The speeds obtained by USB are 682 op/s for

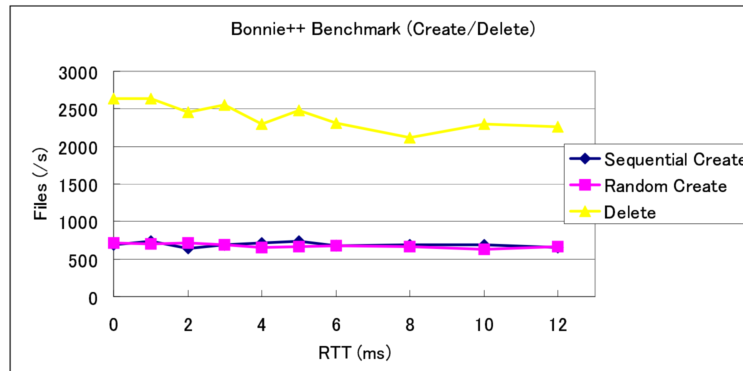

Figure 3.21: Create/Delete Speed

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.

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. A USB camera (which uses the OmniVision OV511 chip) and a USB Audio Class speaker have been tested. These devices work completely transparently using USB/IP on a LAN. Users were able to demonstrate video capture from the camera and successfully played music using the speaker system.

The Linux USB Audio Class driver employs multi-buffering by submitting 2 URBs simultaneously, where each URB is responsible for 5 ms of transactions. Equation (3.2) shows this driver will work with a remote USB audio device provided that the network delay is 5 ms 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.

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 a test LAN, users 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 10 ms. 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 10 ms. Under large network delays, it is possible that the device endpoint buffer may overflow. When network delays approach 100 ms, 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.

3.4.3 Performance Evaluation under IPSec

Support for authentication and security schemes is sometimes an important part for practical applications. As most of USB/IP is currently implemented in the kernel (to avoid memory copy overhead), it is logical to employ an existing kernel-based security mechanism. IPSec [56], 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.

Figures 3.22 and 3.23 show the results of the I/O benchmark described in Section 3.4.2, 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 3.4.1, may also be applied to determine the likely impact of IPSec overhead on the operation of

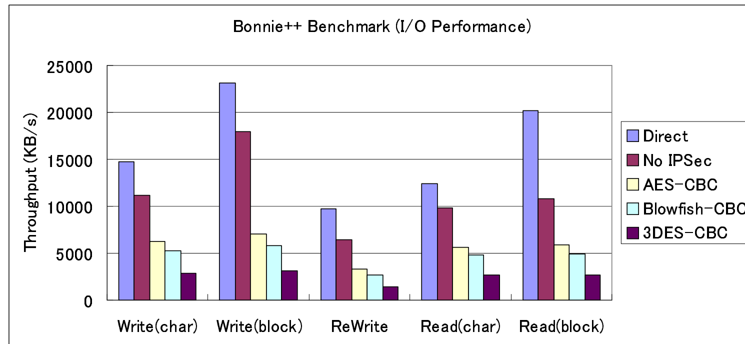

Figure 3.22: Sequential Read/Write Throughput with IPSec

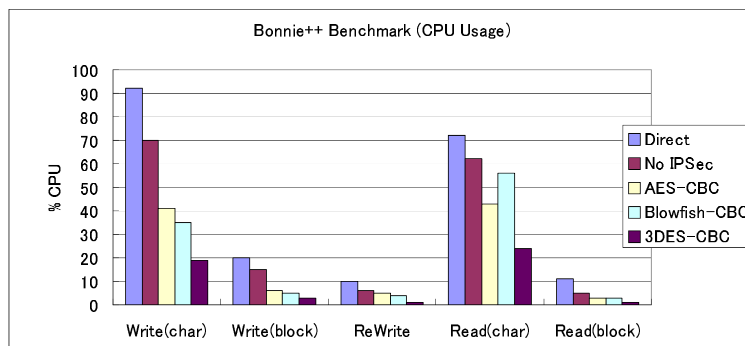

Figure 3.23: Sequential Read/Write CPU Usage with IPSec

USB/IP. Its security mechanism is dealt with more fully in actual applications.

3.5 Discussion

This section discusses experiences and considerations through the design and implementation of the proposed mechanism and its actual usages.

3.5.1 Request Semantics

As for the request semantics discussed in Section 3.2.1 and 3.3.4, the USB model and USB implementations need to be ameliorated also for supporting USB/IP.

Time Constraint

All USB devices tested in the LAN worked correctly without modifying timeout values in drivers; USB implementations use timeout values with sufficient margins for physically-attached devices. However, these constant timeout values defined according to the USB model cannot fit all network conditions with larger latencies than the tested LAN. Timeout values in drivers, including drivers higher than USB drivers, should be adjusted for a target network, and optionally determined dynamically for its network condition. In addition, the request queueing of USB and upper drivers also needs to be adjusted according the performance criteria described in Section 3.4.

Scheduling

Since the prototype did not employ advanced scheduling mechanisms found in USB implementations, periodical transfers were sometimes affected by other asynchronous transfers to a server host, even though the sufficient amount of request queuing was conducted; the playback quality of a USB camera sometimes became worse when a remote USB HDD on the same host was accessed. On the server host, requests are arranged to a host controller according to the scheduling policy of the USB stack. However, the receiving and sending of requests for remote hosts are now performed by kernel threads, which are equally run by a process scheduler. In addition, network packets are also queued equally regardless of USB

transfer types. An analysis of system behaviours in detail and improvements of the current implementation are considered as open issues to be addressed in future work.

Bandwidth

USB driver stacks are differently implemented regarding their bandwidth management of host controllers, since the USB model does not define its design details. Linux kernel 2.6 reserves necessary bandwidth when a periodical request is submitted, Microsoft Windows first reserves required bandwidth when a device is attached, and BSD kernels does not yet fully have this feature. In addition, a USB PDD can dynamically select a suitable device configuration in which necessary bandwidth is narrower, when an application requests it, or the USB PDD detects overcommits or excessive latencies of requests.

In the prototype, some USB devices keep working correctly when network bandwidth is narrowed by NIST Net, because of the "robustness" of drivers and devices; for instance, a driver for a USB camera automatically changes its configuration to capture a smaller screen size, and a USB DVD recorder successfully finishes recording with a protection mechanism on hardware. Since USB implementations sometimes cannot guarantee strict I/O scheduling, most device drivers work under unstable transfer conditions. [10]

However, to extend the USB model and its implementations more appropriately over networks, the following criteria regarding bandwidth management are summarized:

- A USB device should have several configurations[11] with different amounts of bandwidth. Basically, this configuration mechanism for physically-attached devices is considered applicable to USB/IP, which allows a USB PDD to select an appropriate configuration for available bandwidth and user requests, as well as to enqueue sufficient requests against a network latency.
- A USB device with asynchronous transfers should be designed to work cor-

[10] The major part of I/O scheduling is performed by software. There is the possibility that the scheduling is disordered because of various causes (e.g., system overload and inferior hardware).

[11] This means configurations and alternative interfaces in USB terms.

rectly without sufficient bandwidth. This is not specific to USB/IP; since, in the USB model, Bulk transfers are never scheduled if no bandwidth is available.

- If a bandwidth reservation mechanism works in a target network, a USB driver stack should reserve necessary bandwidth when a device is attached. In addition, if possible, a USB driver stack allows a USB PDD to query available bandwidth and network latency via a procedure call, so that the USB PDD can select an appropriate configuration and request queueing.

The design of USB/IP is based on the idea that as many of its mechanisms are implemented in end nodes as possible, thereby enabling them to be used in existing networks without any intelligent Ethernet switch and IP router. The experiments showed, as far as appropriate request queueing was performed by drivers, that most USB devices were correctly operative over IP networks. However, a network QoS technology, if available, will more greatly improve the stability and performance of USB/IP by its bandwidth reservation and priority packet queuing.

Request Cancellation

For the prototype, no problem has been found regarding the cancellation of a request in usages; since it occurs very frequently when stopping or changing transfers, without being implemented correctly it could not be used in actual usages.[12] However, it still does not complete the semantics of the Linux USB stack; in the USB stack, when an endpoint is halted because of an error of a request, a USB HCD must stop processing the request queue of the endpoint, and a USB PDD should retire all pending requests in the queue. And, any further requests to the endpoint need to be rejected at their submission to the driver stack. However, the VHCI driver of the prototype may accept such further requests, since the error of an endpoint is notified of a VHCI driver with a network latency.

In most cases, this does not result in a harmful behavior; in the server side,

[12] This was one of the most difficult parts in the prototype implementation.

these requests (even if newly submitted to the halted endpoint)[13] are rejected by the driver stack with an error code, and the requests are not actually performed in a host controller. However, there may be a possibility that this semantics causes a problem in some circumstances.

Regarding this point, a USB stack should be designed as follows:

- A USB HCD needs to support only asynchronous canceling of a request; if the function call of request cancel is defined as a synchronous call (i.e., search pending requests and unlink a request without any sleeping), it is impossible to implement the function call for canceling a pending request in a server side, since it needs to sleep for waiting the result of unlinking in the server side.

- One of the possible approaches to consistent semantics is considered as follows: A USB HCD accepts new requests to a halted endpoint, but it only appends them to its stopped queue of the endpoint and waits for their unlink requests from a USB PDD. A USB PDD has the same semantics.

  This approach resolves the latency issue regarding the notification of an error by consistent semantics; the requests that reach a Stub driver after being notified of an error are just linked to the stopped queue, in the same manner as pending requests submitted before being notified of the error. All requests in the queue are retired by receiving their unlink request from a USB PDD via a VHCI driver.

The semantics of request queuing and its cancellation are implementation-dependent in detail, so that more investigation is required on this point when porting the prototype to other operating systems.

3.5.2 Error Recovery

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

[13] In another possible design, the Stub driver keeps the requests but does not submit them to the driver stack; it emulates the stopping of the queue for them and waits for their unlinking requests from the VHCI driver.

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 suddenly, 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 aggressive transport scheme is required.

If the USB model is extended to be more suitable for USB/IP, the following alterations are presented:

- Idempotent requests for devices are marked by drivers (e.g., defined by specifications), so that their retries can be possible in an extension mechanism. For instance, most requests reading embedded descriptors are idempotent for devices. If their transfers fail, these requests can be retransmitted by an extension mechanism, without notifying drivers of their errors.

  Moreover, their results become cached in a VHCI driver to improve performance, thereby enabling it to quickly complete the same requests without sending them to remote hosts.

- In addition, the USB model defines more detailed guidelines about implementations of a driver stack to clarify what requests are idempotent for remote operating systems.

The suspend and resume functionality of the USB model are considered applicable, in order to preserve a USB connection even when networks become unreachable within the semantics of the USB model.

3.5.3 Interoperability

Although operating systems have similar USB driver stacks, their structure and semantics are different in detail as mentioned so far. However, an interoperable implementation in other operating systems is basically possible and now ongoing in the open source community of USB/IP. Special care should be devoted to

implementation-dependent parts of the USB model; for instance, queuing semantics in both USB PDD and HCD interfaces, special requests that update status in kernel, and bandwidth allocation mechanisms[14], as well as driver implementation rules such as necessary locking and execution contexts.

A userland implementation of a Stub driver or part of a VHCI driver is possible with an extension of a USB stack; most operating systems support programming interfaces for userland USB drivers to a degree, such as `usbfs` in Linux and `ugen` in BSD kernels. However, these interfaces are less powerful to implement a Stub driver, since they currently lack asynchronous request queueing and isochronous transfers. In addition, some device status maintained in kernel cannot be possibly updated through these interfaces. A memory copy between userland and kernel will involve performance degradation.

3.5.4 Dependability

In the proposed mechanism, all access to remote devices is first processed in device drivers in the same manner as physically-attached devices. Device drivers must be designed to be robust against faulty devices. If a faulty device is attached, an operating system needs to keep working correctly against its harmful behavior. Recoverable driver designs have been studied in order to enable a kernel to recover a driver fault dynamically. For instance, Nooks [94, 93] recovers and rolls back isolated drivers by a protect execution technique and a shadow copy of the driver state. It works in a monolithic operating system. MINIX 3 [43] has explicit recovering mechanisms from critical driver failures by a reincarnation server and a backup data store. It isolates drivers by its microkernel architecture. However, it is difficult to apply these mechanisms to a virtual bus driver. The virtual bus driver does not know the contents of I/O requests and the generalization of rollback is difficult for all types of devices. In a monolithic operating system, since device drivers are deeply coupled with each other, the restarting of parts of drivers is basically difficult without large modification of its kernel. As another approach, it is possible that a virtual machine technology be applied for isolating I/O paths to remote devices. There are some studies to improve driver dependability by a virtual machine monitor. The virtual machine monitor based

[14] This feature may not be implemented fully in an operating system, such as BSD kernels.

on the L4 microkernel allows distinct device drivers to reside in separate virtual machines [64]. An isolation technique used in Xen enables unmodified device drivers to be executed in driver-specific virtual machines [35].

3.5.5 Naming

In remote device technologies, great care has been devoted to achieving flexible resource naming in both local and remote hosts. Previous research [100, 67] has pointed out that naming and I/O mechanisms should be decoupled as much as possible for flexible naming management. The proposed mechanism separates naming and I/O essentially; while I/O messages are common for all hosts no matter how operating systems work, each host preserves its existing naming scheme even for virtually-attached devices. An operating system can assign names to remote devices in its own way, regardless of what names are assigned to the devices in remote hosts. An example of remote device naming in an operating system is presented in a later chapter. Remote device naming in networks can be designed by application developers for their purpose and target environments, considering the guarantee of uniqueness for devices that may be attached and detached physically.

3.6 Related Work

This section notes related technologies of USB/IP.

3.6.1 iSCSI

iSCSI [85] 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). iSCSI is the extension of a SCSI bus over an IP network, and the protocol has been designed to ensure network transparency and interoperability. 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. However, iSCSI is more suitable for storage access than USB/IP because

its I/O granularity is larger than that of a USB storage driver. Recent iSCSI studies, exploring complex interaction between storage and networking drivers, will share performance analysis methodologies and improvement schemes with USB/IP, such as TCP congestion controls and transport encryption mechanisms.

3.6.2 Software and Hardware for USB Device Sharing

Network-enabled USB hubs are developed by companies like Digi, Keyspan, and Silex. These hubs employ proprietary USB extension technologies and provide remote access to USB devices attached to ports on the hub, though in a somewhat limited manner. For example, Digi's AnywhereUSB [52] supports only USB Bulk and Interrupt devices operating at 12 Mbps in a LAN environment. Most USB storage devices, which operate at 480 Mbps, and USB isochronous devices are not supported. Proprietary software products of remote USB access are also developed by Intellidriver, FabulaTech, and Eltima Software. In this chapter, the I/O model of USB/IP has been presented through the experiments. It shows optimization strategies that will enable USB/IP to operate effectively even under larger network delays. The evaluation has shown that, in LAN environments, all the tested devices work perfectly with the original device drivers. USB/IP supports all types of USB devices operating at up to 480 Mbps. It can be said that the academic paper [45, 46] and the open source code of USB/IP contributed to some of these proprietary products.

3.6.3 KVM Switch and NAS

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 potential for useful applications. A practical application of the USB/IP technology is now in progress to share a device more easily between computers.

There are a number of network appliances which export the resources of specific USB devices to an IP network. Some NAS appliances share an attached USB storage device via NFS or CIFS. In addition, some home routers have USB ports that can be used to share connected USB printers or USB webcams. These appliances, which employ abstracted protocols[15], do not support the low-level operations which a required to allow the remote devices to operate in a transparent manner.

3.6.4 Wireless USB

A new technology under development is Wireless USB [3], 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.

3.7 Summary

In this chapter, a network extension mechanism of USB was proposed, which allowed operating systems interoperable remote access to peripheral devices with full functionality. USB, a multipurpose peripheral interface based on host controller hardware, enables operating systems to decouple messaging to their devices from bus controls inside its kernels; messaging to devices, which is essentially based on its peripheral bus model, is independent of operating systems, so that the proposed mechanism aims to exploit it also for transparent remote device access among different operating systems. In the proposed mechanism, peripheral devices are never abstracted for remote access and raw device hardware is exported over IP networks to target computers. However, USB is not originally designed to be extended as the proposed mechanism; this study needed to explore its possibility and limitation against its peripheral bus model and their

[15] Operations specific to device hardware do not tended to be supported.

implementations.

USB/IP was designed and implemented carefully to meet full functionality, network transparency, interoperability, and generality. It allowed a range of remote USB devices to be used from existing applications without any modification of the application or device drivers. A range of experiments was undertaken to establish that the I/O performance of remote USB devices connected using USB/IP was sufficient for actual use. The performance characteristics of USB/IP were also determined, and optimization criteria for IP networks were developed.

There are three primary design criteria related to performance 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 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.

Moreover, through experiments and considerations of its development and actual usages, additional design criteria were raised from request semantics differences between local and remote device access; especially about time constraints, scheduling, bandwidth, synchronization, and cancellations of requests. As in discussions, USB/IP implementations need to devote special care and mechanisms to these points. Also, the USB model and its implementations need to be extended for more suitable support of the network extension.

Chapter 4

A Device Sharing System for USB/IP

A device sharing system among computers has been developed as one of the applications of USB/IP. As discussed in the previous chapters, the proposed mechanism extends device connections virtually over IP networks without any abstraction. Operating systems can access remote devices as they are. It has significant advantages for full functionality, network transparency, generality, and heterogeneity. On the other hand, other remote device technologies in the software level abstract a device to resources suitable for sharing device-derived functions over networks. These abstractions multiplex hardware access from computers in the resource-provider side. In USB/IP, the multiplexing of hardware access is never offered by itself; only a client operating system is responsible for it. This chapter proposes a device-sharing system based on USB/IP that provides users with seamless device access the same as switching peripheral cables among computers. It is intended to show how a device and device-derived resources are presented to users as well as how the device-sharing system is modeled in a consistent manner with existing device management of operating systems.

4.1 Device Sharing Based on USB/IP

USB/IP can be a reconfigurable device access technology in distributed systems, which enables computers to change peripheral topologies by request virtually

regardless of their physical locations. I/O requests on a peripheral bus are transferred over IP networks to redirect raw device access between computers. A virtual bus driver implemented as a peripheral bus driver emulates remote device attachment in the lowest level of kernel, so that most drivers and applications can also work for remote devices without any modification as if they were directly attached. It can basically support all types of devices on a peripheral bus and can be applied to most operating systems and computers.

One of the possible applications of USB/IP is a device-sharing system among computers. Users can share their peripheral devices with each other through IP networks without any physical reattachment of devices, like a peripheral cable is virtually passed to another person by request. A more advanced sharing system enables users to gather devices from other computers regardless of physical locations and to set up a custom-made computing environment on demand anywhere. Various device sharing systems can be developed by utilizing peripheral bus extension in combination with other distributed technologies.

To discuss essential technical elements for these systems, the target of a device-sharing system is focused on casual device sharing between users in mind. It is intended to show what technical components are required for such systems and how devices are presented to computers and users through management techniques among computers and also inside a single computer.

4.2 Requirements

The fundamental and distinctive requirements for USB/IP device-sharing systems are discussed here.

4.2.1 Location-Independent Handling

The location-independent handling of peripheral devices is required for seamless device sharing. Whether a device is attached physically or virtually, most drivers and applications never see the difference between two types of attachment. Once connected, a physical peripheral topology is concealed inside bus drivers. However, before connected, a user is still bothered about the location of a target device. If the user wants to use a remote device attached to another computer,

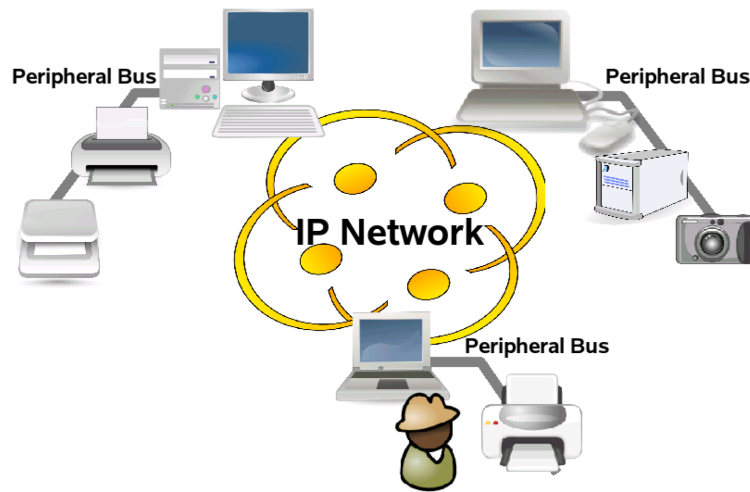

Figure 4.1: Device View without USB/IP

he needs to set up a new virtual connection for it. USB/IP itself enables only remote device access from computers as shown in Figures 4.1 and 4.2. Users still need to be concerned about the distribution of the physical bus.

Instead, the device sharing system should be able to provide flat device views of users through which both remote and local devices are handled in the same manner (Figure 4.3). The flat device handling allows operating systems to customize the device views of users according to its management policy, regardless of physical locations. For example, the system presents an integrated virtual bus that allows access to any device on demand or by request for a user.

This separation of the methods of device access and presentation is a fundamental mechanism for advanced sharing features. For instance, first, each user can have his original scope of devices beyond networks. It contains both remote and local devices depending on conditions. A context-aware presentation is also possible by reflecting the attributes of devices and users. A sharing system can dynamically show a user only available devices in the room where he exists now. It can be developed with a directory server or a group communication technology such as multicast and peer-to-peer. Second, a session can be defined to abstract a series of device connections for a device usage. A pairing of a device and a computer can be recorded in a session server, so that virtual or physical connec-

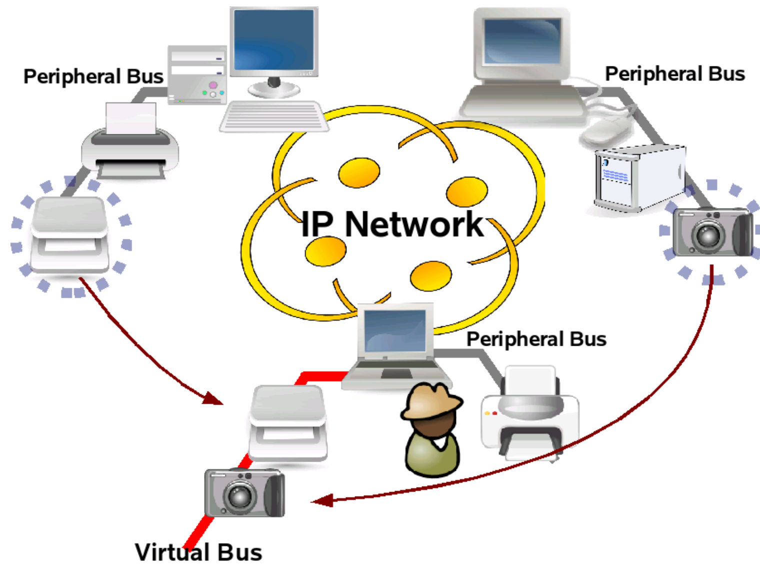

Figure 4.2: Device View with USB/IP

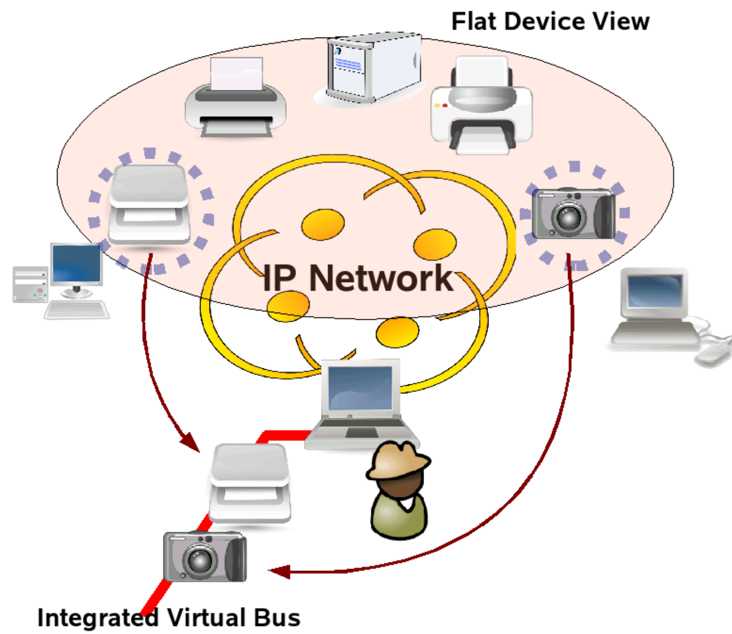

Figure 4.3: Flat Device View for Device Sharing

tions between them are always established automatically. This enables persistent device usages even though a device or a computer is moved to another place. It is also possible that a pairing of a user and devices is maintained for a working set of devices. The user obtains the same computing environment at any computer.

4.2.2 Device Ownership and Control

The device sharing on the system needs to be a straightforward extension of the physical device usages. Actually, all devices are sometimes owned by a user or a group of users. Since an owner of a device is responsible for its usages, the owner can decide whether or not others can use his shared device and when they should stop using it. Though devices in the system are handled in the same manner regardless of locations, the owners' controls over devices still need to be preserved in conformity to physical topologies and their policy. The system needs to address the consistency of device sharing between actual and virtual peripheral topologies as well as the flexibility of usages.

4.2.3 Device Management Redesign

The device management of operating systems should be redesigned to support device sharing based on USB/IP. A USB/IP device-sharing system shares not only device-derived resources but also device connections themselves among computers. An operating system has resource management mechanisms for processes and users. However, such mechanisms cannot fully address the issues related to sharing of device connections because they are designed for only resources made from already connected devices.

Existing device management mechanisms lack the flexible control of device attachment. A system policy should be able to allow or disallow the device attachment driven by entities on an operating system, so that non-privileged entities can control a device flexibly as far as the system policy allows. Moreover, the system must work correctly when it allows non-privilege device controls. In most operating systems, the attachment of a device and its resource allocation are intended to be performed by only administrative privileges. This limitation is also imposed to virtual attachment by USB/IP. Since there is no access con-

trol mechanism for device attachment, users cannot import and export devices without administrative privileges.

The system needs some arbitration mechanisms to share device attachment among computers. Device attachment is basically considered as an atomic resource that cannot accept concurrent access from multiple computers. It requires exclusive access among entities inside and outside a computer.

4.3 Design

This section summarizes the key idea of the device sharing system based on USB/IP.

4.3.1 Device Presentation

In order to present the location-independent handling of devices in a straightforward manner from the real world, the system introduces *a device object* with three operations as a basic object into fundamental device management. It corresponds to a device attached to a computer, which is a straight presentation of a basic unit of resources shared in the system. A device object is manipulated through its operations by *a subject* on the system. A subject is a basic unit that can obtain a device object and use the resources derived from it. It corresponds to a user, a process, or a computer; these are entities on an operating system, which may get a device connection and control it exclusively (Figures 4.4 and 4.5).

A device object is exclusively assigned to one subject. However, this relationship is changed dynamically by operations to device objects in the system. A subject can manipulate a device object through three operations as follows:

Acquire A subject requests a device object. When a subject invokes this operation, the system decides whether the operation is allowed or not. If allowed, the operation succeeds. The device is newly assigned to the subject, and the previous subject that had the device gives it back. Otherwise, the operation fails.

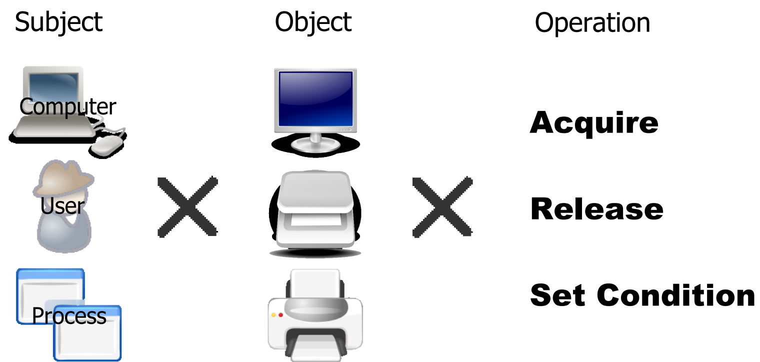

Figure 4.4: Device Object Operations

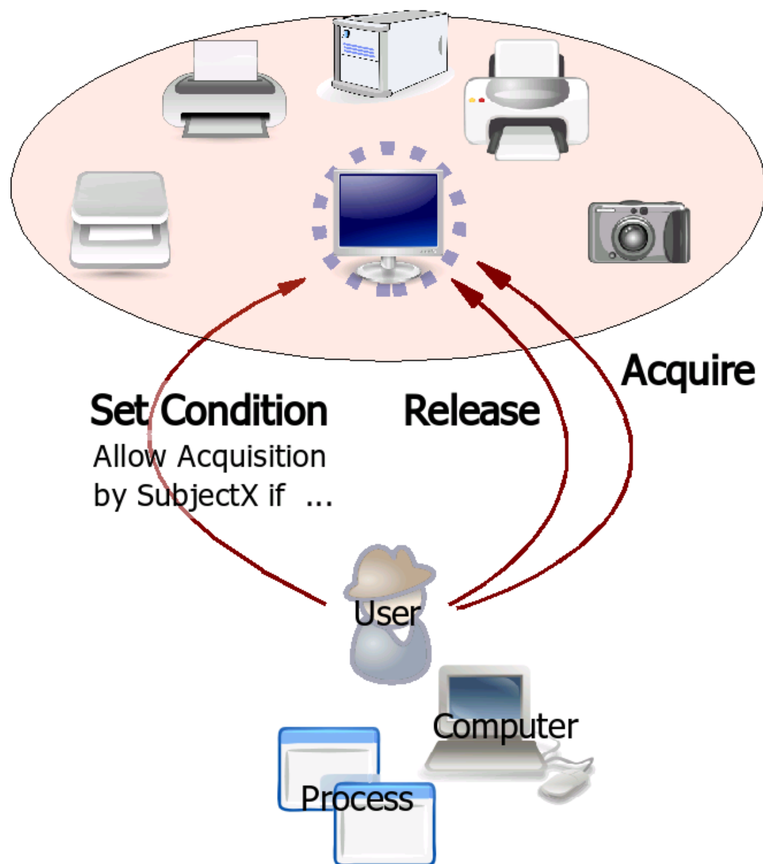

Figure 4.5: Device Object and Flat Device View

Release A subject releases a device object if the subject has it. Then, the system assigns it to another subject.

Set Condition A subject can make an attempt to set permission for other subjects to get the device. The permission is composed of condition descriptions that define how the subject tries to allow other subjects' acquisition of devices. It is described by parameter values that show attributes of devices and subjects. The permissions defined by subjects are incorporated into the device assignment decision that the system makes.[1]

Exclusive Access

The system splits device management into two levels; one is device hardware attachment, and the other is device-derived resources from it. The former is controlled by the system, and the latter by a subject. This enables the straightforward abstraction of device hardware for exclusive access management as well as the flexibility of device usages. A device object is assigned to one subject exclusively. This decision is performed by the system according to permissions. Once the device is assigned to the subject, the subject is fully responsible for the control of the device; the subject can abstract the device for use. It can bind device drivers and use resources derived from it. It can also export the resources to share them. The system is responsible for exclusive access of raw device hardware [2]. However, device-derived resources can be concurrently accessed as the subject intended. To implement this idea, the device management mechanism of an operating system needs to be changed as described in a later section.

Condition Description

The system provides subjects with flexible condition description, so that distributed devices are managed efficiently under various situations. An acquisition

[1] Actually, this operation is considered as a sort of delegation of the privilege of getting a device object exclusively. Section 4.3.2 discusses this point with the privilege management of device usages on an operating system.

[2] Not raw device access in UNIX.

condition is described by the following parameters. The system evaluates conditions and decides the assignment of a device object. This procedure is described in Section 4.3.2. A subject can put a priority value into a condition to exercise a management policy with its strength.

Subject Attributes Parameters that specify a subject or a group of subjects. For example, the system allows "computers in a room" to acquire a shared device.

Device Attributes Parameters that specify a device object or a group of device objects. For instance, a user wants to specify "a keyboard on my desk" to share it among computers.

Current Status Parameters that reflect current status of an assignment. For instance, an elapsed time value enables a subject to give a device to another subject with a time limit.

Priority A value to set priority of a condition for flexibility. It is also used for privilege delegation in the device management of an operating system as in Section 4.3.2.

4.3.2 Device Management Model

The overview of device management on an operating system is illustrated in Figure 4.6. A domain is a key conceptual mechanism that loosely isolates a raw device and its derived-resources from other subjects, so that a subject can be fully responsible for a device object. A subject can bind drivers and share resources with other subjects. Other subjects cannot access them without the arbitrary control by the subject. A domain also works for preventing the subject from violating other privileged resources, and enables non-privileged entities to control device attachment flexibly as the system allows.

The device management on an operating system is divided into two parts; one part is responsible for device attachment by the system, and another is for resource management by a subject. Because of this decoupling, the system can handle the exporting of a device to another computer in the same way as the providing of the device to a local subject on it. Also, exclusive access is achieved

flexibly; while a device is exclusively assigned to a subject according to a system policy and its request, device-derived resources can be shared as the subject intends.

System Behavior

When a subject requests the acquisition of a device, the system starts to prepare the target device, which may involve a new virtual connection to another computer. After the device becomes available on the system, it checks whether this request is approved or not, according to available conditions with subject attributes, device attributes, and current status. If approved, the device is provided to the subject and then the subject manages it. In the case that the subject is another computer, the system exports the device to it. If disapproved, this attachment is canceled; a virtual connection is dropped or a physical connection is ignored.

When a device is attached to a computer without any request, the system checks whether the device can be connected or not in the same way as the above with a default policy; the computer is a subject for assignment decision.

Delegation Chain

Conditions are evaluated as a delegation chain to the subject that requests the acquisition of a device, so that the system can preserve existing privilege management on an operating system as much as possible. For example, an administrative user or process needs to control device usages on the system more strongly than normal users. As far as an administrative user allows, a normal user can utilize devices and export them.

A more administrative subject on an operating system has stronger privileges for devices. It gives its privileges to less administrative subjects conditionally according to condition descriptions by invoking Set Condition operations of device objects. If a subject has a privilege of a device, it can utilize the device exclusively, give the privilege to other less administrative subjects conditionally, and cancel the device assignment of less administrative subjects for taking it back.

Figure 4.7 shows how this mechanism works. Each subject can invoke Set condition operation to allow other subjects to acquire devices. When a subject

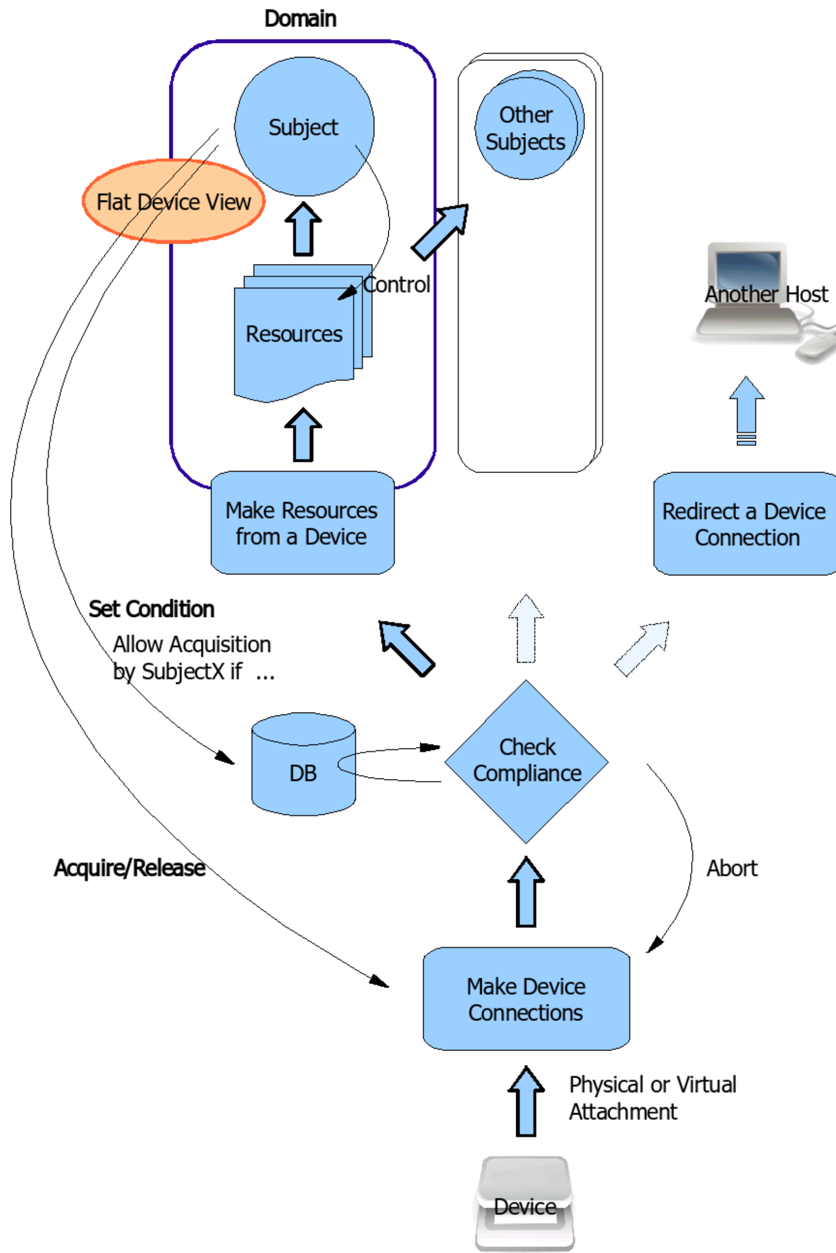

Figure 4.6: Device Management Model

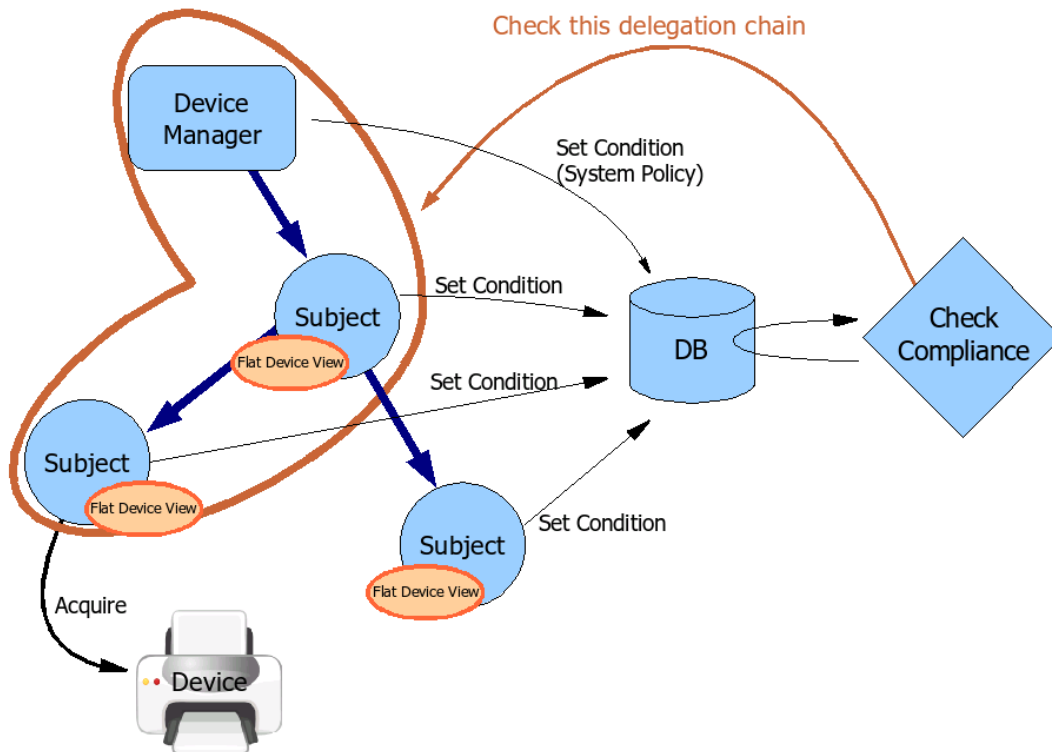

Figure 4.7: Delegation in Device Management

invokes the Acquire operation of a device object, the system evaluates all conditions on delegation paths from the most administrative subject to the target subject.

Figure 4.8 is an actual example that illustrates delegation flows between two computers. User F succeeds in the acquisition of Device X exported from Computer P. In this case, both computers have individual device management mechanisms. In Computer P, Device X was requested for acquisition by Device Manager D in Computer Q. To check whether this request was approved or not, Computer P evaluated condition descriptions on the delegation chain to Device Manager D. In Computer Q, condition descriptions from Device Manager D to User F were evaluated as a delegation chain. Consequently, Computer P checked whether its device could be exported to Computer Q, and Computer Q checked whether the device could be imported for User F.

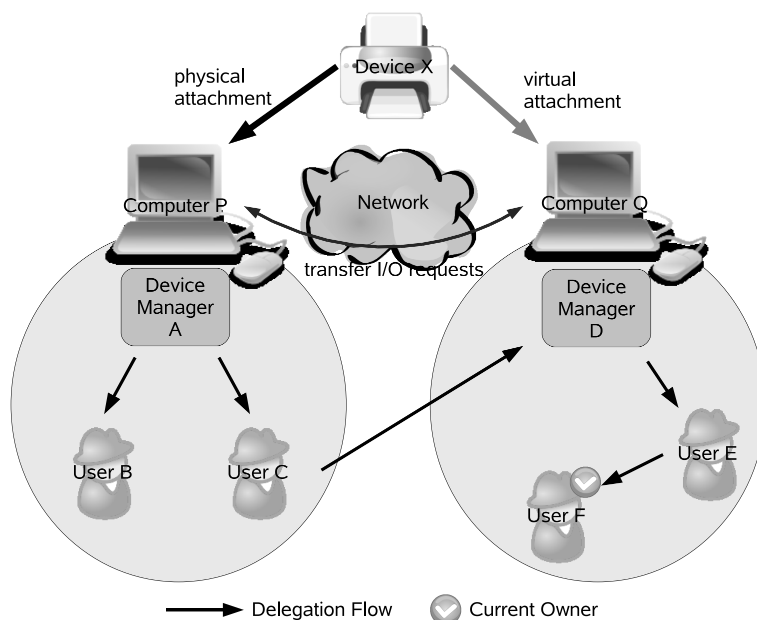

Figure 4.8: Delegation between Two Computers

4.4 Implementation

A prototype system of device sharing based on USB/IP was implemented.

4.4.1 Strategy

Device Attachment Uniqueness

In order to generate a unique device object identifier, the system identifies a device object by its attachment, not by its hardware itself, since an operating system lacks a general method to identify each peripheral device. A unique serial number is not embedded in most consumer devices. Even if embedded, there is no guarantee of its uniqueness. Also, most devices lack a nonvolatile memory area, to which a computer writes a generated identifier. For instance, if a device is unplugged from a computer and plugged in again, the computer does not understand that the same device is attached again in a general way.

A device object identifier on the system is composed of three parts: a host identifier, a bus topology identifier, and the time when it is attached to the

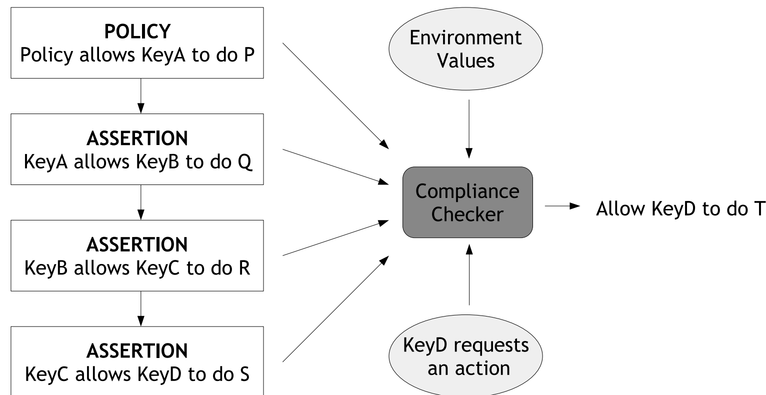

Figure 4.9: Trust Management Concept

host. The device management mechanism of an operating system is modified to generate device identifiers correctly, recording device attachment time, so that it becomes globally unique to differentiate device objects.

Trust Management

Trust management is utilized to implement privilege delegation in device management on operating systems. Trust Management, first introduced by PolicyMaker [10], is an authorization management framework that makes the decision to allow a principal (e.g., user or application) to do something. While X.509 certificates and PGP bind a public key with a principal's name, trust management binds a public key directly with an operation that a principal is permitted to do. A principal issues a new assertion[3] and signs it by its private key, so that it can delegate part of its authority to other principals[4]. As figured in Figure 4.9, since an assertion includes an authorizer and licensees specified by their public keys, a trust management system can confirm delegation chains without finding trust paths of public keys. Authorized operations in an assertion (P, Q, R, and S) are described in a machine-readable form. Basically, a compliance checker and an assertion language are provided to develop applications of trust management easily.

[3] A formatted text that describes the operations principals are allowed to perform.

[4] An authorizer still has the same authority.

Though there are other trust management systems (e.g., SPKI [29], REFEREE [21], QCM [39], and SD3 [55]), a prototype implementation of the proposed system utilizes KeyNote [9, 11]. KeyNote defines the syntax of assertions and provides a compliance checker with signature verification. Its syntax meets assertion monotonicity; deleting an assertion never causes a disapproved action to become approved. The result of a compliance check is a compliance value that presents how a request should be handled by an application; this corresponds to the priority of a delegation chain. It is used as an authorization mechanism in some applications [13]; for example, a distributed file system [71], an IPSec policy control mechanism [12], and a distributed firewall [54]. The benefits of trust management for the prototype system are summarized as follows:

- Developing a new delegation mechanism for the system is avoided.
- Any subject can issue a new assertion anytime only with the target public key in a decentralized manner. This action corresponds to the Set Condition.
- Each operating system can enforce its system policy independently.
- An electrically-signed assertion can be transferred via non-encrypted transport and stored to non-encrypted storage.
- Lacking an assertion never causes the increase of a compliance value. Even though the system fails to collect all related assertions, the result does not violate a system policy.

In the prototype system, a subject prepares other subjects' public keys by other mechanisms, such as PKI, PGP's web of trust, or out-of-band.

Resource Isolation

The concept of a domain can be implemented in several ways depending on the degree of resource isolation and the feasibility against commodity operating systems. [5] A design issue is raised by the fact that most operating systems lack

[5]This discussion relates to the system dependability of peripheral bus extension noted in Section 3.5.4.

consistent handling of device-derived resources. An operating system abstracts a device into to various resources in multiple levels. However, device drivers provide different resource interfaces and interact with each other complexly. The total resource isolation from the lowest level is difficult to resolve.

One possible approach is to exploit driver separation mechanisms for commodity operating systems [93, 94, 35, 64, 37]. They separate driver execution environments by a protect domain, a virtual machine monitor, or a micro-kernel system. These mechanisms require driver-specific modifications from all drivers on I/O paths.

Another approach is to utilize virtual machine mechanisms [6, 27, 91], thereby allowing an independent system for a domain. For example, UML [27], which runs a Linux kernel as userland processes, enables the system to isolate all driver behaviors derived from a device into a virtual machine. It is basically possible to create a virtual machine for a domain by developing a userland management mechanism without large modifications of kernel code.

In contrast, more passive approaches are also possible based on the idea that matured operating systems are implemented carefully to be dependable; a device I/O is basically separated from others in a single kernel, and only the interfaces to resources should be managed for isolation. The system separates resource name space for each domain and makes up for access control to resources. A kernel extension mechanism [101] for security policy enforcement [66] will be exploited for customizing system calls to device-derived resources.

The prototype system, developed for Linux, allows the private name space of device files for each subject as a domain. Though this does not support the concept of a domain completely for various devices and abstractions, it is intended to be focused on the design discussions by demonstrating the first usable prototype.

4.4.2 Overview

The overview of the device management mechanism is illustrated in Figure 4.10. It utilizes the hot plugging support of an operating system and avoids the modification of its kernel as much as possible. USB/IP is employed as one of the remote device technologies for it. Its transport is encrypted by TLS [26] and IPSec [56] by request. For the prototype implementation, Multicast DNS [19] is exploited

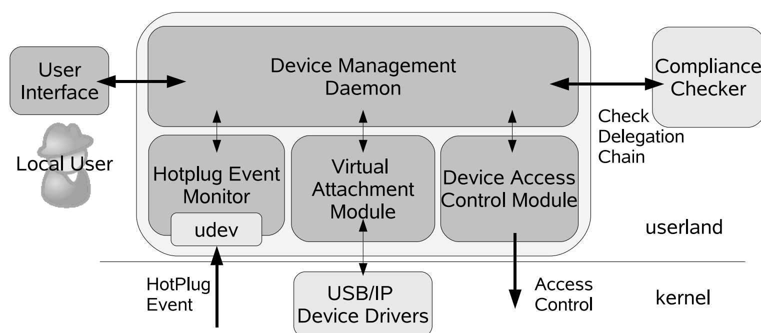

Figure 4.10: Design Overview

for a flat device view of a user. As a Set Condition operation, a user issues an assertion of KeyNote and registers it to the system. A user also unregisters it for revocation.

The system works in the same manner for both Acquire requests from a local and remote subject as illustrated in Figures 4.11 and 4.12. When the system gets the request from a local subject, it collects subject information (e.g., its public key, UID, and GID) and device object information (e.g., device object identifier, the public key of its exporting host, retrieved device attributes, and its status), and then evaluates delegation chains with available assertions and a system policy. If approved, the system sets up its private device files for the subject. All device files derived from the device are created only in the subject's namespace.

When the system gets the request from a remote subject, it also collects subject information, device object information, then checks delegation chains. Since this case means the subject is the other computer requesting Acquire, subject information includes its IP address and its public key. If approved, the device is exported to it. Then, the system of the other computer will also check whether the Acquire operation from its local subject is approved or not.

Assertion parameters are defined for the system, which are based on the discussion in Section 4.3.1. (See Table A.1 in Appendix.) An example assertion is illustrated in Figure 4.13. The condition description allows that peripheral devices used for presentation are temporarily leased to the mobile computer of a guest. As far as the mobile computer is connected to a local area network of the `naist.jp` domain, it can temporarily connect the VGA device (`0711:0902`) and

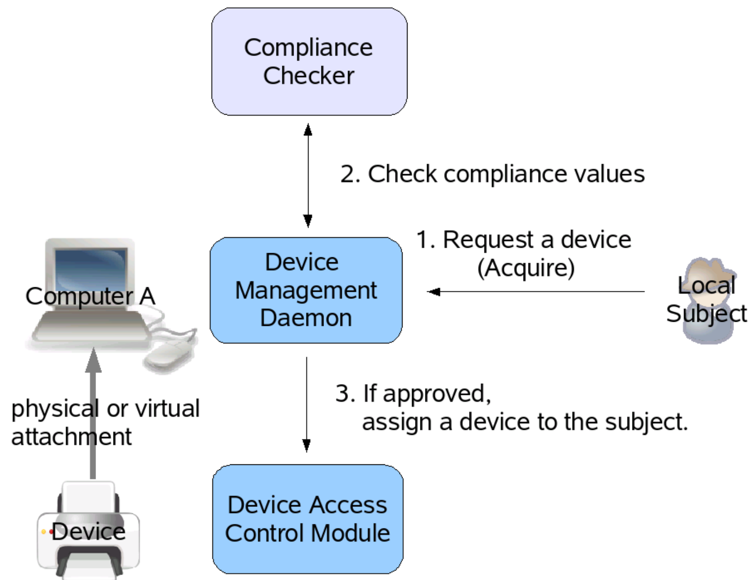

Figure 4.11: System Behavior (Local Request)

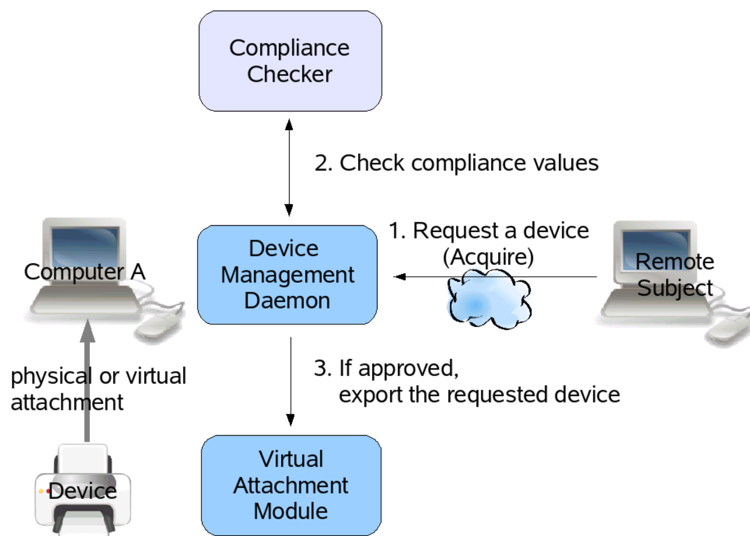

Figure 4.12: System Behavior (Remote Request)

```
KeyNote-Version: 2
Authorizer:
         "rsa-base64:MEgCQQCrXFSoFxVNjUVfd5GqNBFCFEavSlAV9\
         Fzwt45aKg7iyRjoyHcOPb1Pz9NPm8RGMDueiBL9ddnc7Om0J6\
         asb8t9AgMBAAE="
Licensees:
         "rsa-base64:MEgCQQC3uWZVBVPWgVfDcaqNEfNIXkgvPmZpC\
         IA3tK8N/RDpF5Am8pdTciKVUB6AaxVj0COpXAmPl3RNa1pYHb\
         pxsBhLAgMBAAE="
Conditions:
         (app_domain == "DMD") -> {
                  (host ~= "^.*\.naist\.jp$" && devhost == "192.168.0.100") -> {
                           ( (idVendor == "0711" && idProduct == "0902")
                           || (idVendor == "08bb" && idProduct == "2702") ) -> {
                                    ( &lease_time <  3600.0 ) -> "3" ;
                                    ( &lease_time <=    0.0 ) -> "5" ;
                           }
                  }
         }
Signature:
         "sig-rsa-md5-base64:iOkaUN+Teotclq+Hr1p7OE+W8s2xP\
         YTAYtsD91c2PPkeg9ZVxtyfPJOg+x0OH1fZnuASaMDIzttWyn\
         90gpcxnQ=="
```

Figure 4.13: Assertion Example

the speaker device (`08bb:2702`) exported by a device server (`192.168.0.100`). For one hour, the priority value is `3`. After that, the value becomes undefined (minimum), and the server will take back the devices.

4.4.3 Prototype Demonstration

The prototype system was implemented by Ruby 1.8.4 and KeyNote 2.3 on Linux Kernel 2.6.16.5. Hot plugging events and driver binding events from kernel were captured via a userland device event manager (udev v088). The USB/IP implementation was improved to be able to change driver bindings and target subjects dynamically.

The prototype system was tested among computers in a LAN environment. The administrator of each computer defined its system policy that described how locally-attached devices could be assigned to subjects and also how remote devices could be assigned to local subjects. These procedures corresponded to Set Condition by a subject in the device sharing model. Once a device was physically attached to a computer, the computer advertised it to other subjects by exploiting

a Multicast DNS daemon.

Figure 4.14 illustrates a usage example. First, a user listed devices available to him and then tried to acquire the 7th device (OV511+ WebCam). This request was approved by both systems (i.e., exporting and importing sides), and the requested device was attached to the localhost for the user as the 4th device. As the output of `tree` shows, the system created private device files for subjects under the `/dmd/user/` directory. For instance, the subject `root` had one device named as `usb_3-1.1.2_1145544020@10.0.0.3`, which was abstracted to 3 private device files (`bus_usb_006006`, `scd0`, and `sg0`). In the prototype system, changing device assignment required a few seconds at most. The main part of this latency was the initialization process of USB drivers.

A user wrote new assertions for his friends with their public keys and passed the assertions to the licensees by e-mails or an Internet messenger. Then, his friends registered received assertions to the systems of their computers in order to use devices allowed by the user.

The prototype implementation works correctly as intended by the system design. It achieved the concept of the proof of the device sharing system in a primitive way. It does not provide an intelligent device view customized for each subject, a complete resource isolation mechanism, and an advanced user interface. However, these features will be implemented in actual applications optimized for their purposes.

4.5 Discussion

The system exploits KeyNote to implement privilege delegation quickly for the prototype. It greatly reduces implementation cost for it. However, issuing its assertion correctly as intended is difficult for most users; it needs to get a public key of a target, write a condition according to its format, sign it electrically, and pass it to appropriate hosts. This procedure should be simplified in a practical system by sophisticated tools. Another aspect of KeyNote is that a user cannot easily know what authority is currently given to him. It provides no way of knowing except by performing a compliance check with parameters, even though all assertions can be seen.

```
taka@deux:~$ devconfig --list
  6 : usb/3-1.2/2006-04-20T14:06:26Z@10.0.0.4
    : OnSpec Electronic, Inc. (55aa:0201)
  7 : usb/3-1.3/2006-04-20T14:40:14Z@10.0.0.3
    : OmniVision Technologies, Inc. / OV511+ WebCam (05a9:a511)

taka@deux:~$ devconfig --attach 7
complete

root@deux:~# devconfig --list localhost
  2 : usb/6-1/2006-04-20T14:41:05Z@localhost <- usb/3-1.1.2/2006-04-20T14:40:20Z@10.0.0.3
    : I-O Data Device, Inc. (04bb:0201)
  3 : usb/6-2/2006-04-20T14:37:52Z@localhost <- usb/5-4/2006-04-20T14:30:09Z@10.0.0.3
    : Magic Control Technology Corp. (0711:0902)
  4 : usb/6-3/2006-04-20T15:18:05Z@localhost <- usb/3-1.3/2006-04-20T14:40:14Z@10.0.0.3
    : OmniVision Technologies, Inc. / OV511+ WebCam (05a9:a511)
  6 : usb/3-1.1/2006-04-20T14:31:14Z@localhost
    : Alcor Micro Corp. / Keyboard (058f:9410)

root@deux:~# tree /dmd/
/dmd/
|-- host
|   |-- 10.0.0.3
|   |   |-- bus_usb_006_005 -> /dmd/user/taka/usb_5-4_1145543409@10.0.0.3/bus_usb_006_005
|   |   |-- bus_usb_006_006 -> /dmd/user/root/usb_3-1.1.2_1145544020@10.0.0.3/bus_usb_006_006
|   |   |-- bus_usb_006_007 -> /dmd/user/taka/usb_3-1.3_1145544014@10.0.0.3/bus_usb_006_007
|   |   |-- scd0 -> /dmd/user/root/usb_3-1.1.2_1145544020@10.0.0.3/scd0
|   |   |-- sg0 -> /dmd/user/root/usb_3-1.1.2_1145544020@10.0.0.3/sg0
|   |   |-- sisusbvga0 -> /dmd/user/taka/usb_5-4_1145543409@10.0.0.3/sisusbvga0
|   |   `-- video0 -> /dmd/user/taka/usb_3-1.3_1145544014@10.0.0.3/video0
|   `-- localhost
|       `-- input_event3 -> /dmd/user/taka/usb_3-1.1_1145543474@localhost/input_event3
`-- user
    |-- root
    |   |-- all
    |   |   |-- bus_usb_006_006 -> /dmd/user/root/usb_3-1.1.2_1145544020@10.0.0.3/bus_usb_006_006
    |   |   |-- scd0 -> /dmd/user/root/usb_3-1.1.2_1145544020@10.0.0.3/scd0
    |   |   `-- sg0 -> /dmd/user/root/usb_3-1.1.2_1145544020@10.0.0.3/sg0
    |   `-- usb_3-1.1.2_1145544020@10.0.0.3
    |       |-- bus_usb_006_006
    |       |-- scd0
    |       `-- sg0
    `-- taka
        |-- all
        |   |-- bus_usb_006_005 -> /dmd/user/taka/usb_5-4_1145543409@10.0.0.3/bus_usb_006_005
        |   |-- bus_usb_006_007 -> /dmd/user/taka/usb_3-1.3_1145544014@10.0.0.3/bus_usb_006_007
        |   |-- input_event3 -> /dmd/user/taka/usb_3-1.1_1145543474@localhost/input_event3
        |   |-- sisusbvga0 -> /dmd/user/taka/usb_5-4_1145543409@10.0.0.3/sisusbvga0
        |   `-- video0 -> /dmd/user/taka/usb_3-1.3_1145544014@10.0.0.3/video0
        |-- usb_3-1.1_1145543474@localhost
        |   `-- input_event3
        |-- usb_3-1.3_1145544014@10.0.0.3
        |   |-- bus_usb_006_007
        |   `-- video0
        `-- usb_5-4_1145543409@10.0.0.3
            |-- bus_usb_006_005
            `-- sisusbvga0

12 directories, 24 files
```

Figure 4.14: Usage Example

Some USB drivers need to fix their errors in order to avoid kernel panics when dynamically changing their assignments, which seem to be not well-tested regarding detachment in usages. This motivates more aggressive resource isolation as noted in Section 4.4.1. Also, most applications do not handle the disappearance of a device well, sometimes hanging up in the worst case. The useful notification framework of device release is desirable to pursue the appropriate handling of detachment.

4.6 Summary

A device sharing system was presented as one of the applications based on USB/IP. USB/IP enables raw access of remote devices without any abstractions. The multiplexing mechanism of device access is not provided by itself. This feature is very different from other remote device technologies designed to share multiplexable resources derived from a device. Therefore, this chapter was intended to show a fundamental system model for USB/IP device sharing and to give basic design criteria for advanced practical systems. In the system, devices are location-independently managed by three primitive operations and incorporated into existing operating systems by a device-derived resource isolation technique and a privilege delegation mechanism. A prototype system was operated correctly as proof of this concept; users shared devices flexibly among computers with policy enforcement, regardless of their physical locations.

Chapter 5

Open Issue

The hardware and software models of Bluetooth and IEEE1394, which are other multipurpose peripheral interfaces based on host controller hardware, are quite similar to that of USB. These request semantics to an attached device, which are operating-system-independent, are basically applicable to access remote devices without large modifications, by request redirection mechanisms over networks in operating systems.

For the Bluetooth interface, a virtual bus driver can be implemented as a host controller driver. Actually, an emulation driver of a host controller is already available for test purposes, which will be used for developing the Bluetooth virtual bus driver. For the IEEE1394 interface, the CSRs (Control Status Registers) of a host controller interface will be emulated for a virtual host controller driver. Most I/O requests are just redirected to a remote device and some requests related to bus status need to be intercepted and modified for emulation.

Although, basically, the proposed mechanism is considered applicable to them, other considerations are raised from differences of their models and implementations. The resource allocation mechanism of the IEEE1394 model is more aggressive than that of the USB model. A host controller arbitrates bus utilization with other nodes and reserves bandwidth for isochronous transfers. This means its network extension mechanism first acquires necessary bandwidth in both a remote host controller and a network, and then keeps it while a device is used. In addition, multimedia devices and applications of IEEE1394 tend to be more latency-sensitive than those of USB. On the other hand, the USB model allows

implementations to select device modes dynamically against available bandwidth even when a device is in use. This part of its model can also be applicable to the network extension of the proposed mechanism.

A driver stack should not be implemented to support only specific types of host controllers, especially attached to a PCI bus. A virtual bus driver can be developed without emulating PCI bus behavior as found in VMMs, so that its implementation becomes greatly simplified, since it needs to emulate only the logical status of a host controller. In addition, an allocation of a DMA-capable buffer is slightly costly in some architecture. For instance, in an IEEE1394 stack in Linux, class drivers perform PCI-specific DMA operations in several places.

The Bluetooth bus extension was discussed in [51], which bridged separated piconets to access remote Bluetooth devices by TCP/IP. ACL (Asynchronous Connection Less) and SCO (Synchronous Connection Oriented) links were established over a network by emulating the device of the other piconet and redirecting data packets. Its performance degradation was acceptable in the tested network. Also, the IEEE1394 bridging by Ethernet was studied in [58]. These studies showed performance feasibility and bus emulation techniques. Compared with these studies about the bridging of a peripheral bus, the proposed mechanism is focused on seamless remote device access from operating systems. It has a virtual bus driver to integrate remote devices seamlessly. Although these peripheral buses as well as USB, are not originally designed to be extended over IP networks, the experiments showed non-abstracted raw devices could be accessed under today's networking technologies in a feasible manner to a degree.

Chapter 6

Conclusion

This thesis is motivated by the ideal goal of distributed system technologies; users obtain optimized computing environments dynamically composed of distributed hardware resources over networks on-demand and seamlessly. Obviously, a remote device technology is one of the essential keys to this goal, which enables the system to be reconfigurable and adaptable in its device usages for various conditions and purposes. Although many remote device technologies have been studied over the years, there is still lacking a fully general and interoperable device access mechanism over IP networks. This feature is considered necessary to tackle the increasing diversity of devices and computers in the coming future. Most remote device technologies developed in operating systems are based on interfaces and semantics inside their kernels, thereby involving operating-system-dependent structure and semantics in their extension mechanisms.

6.1 Contribution

This dissertation has proposed a network extension mechanism of USB, one of the multipurpose peripheral interfaces by host controller hardware; it decouples request messages and semantics to devices from bus controls, and also exploits them for remote device access in the same manner as internal access to physically-attached devices. Since its remote device mechanism becomes a natural extension of the peripheral bus model, it is basically independent of operating system structure and semantics, and therefore interoperable among different operating

systems.

However, since the peripheral bus model and its implementations are not intended to be extended over IP networks, this research has tackled problems arising from the inevitable semantics differences in the extension mechanism. It has clarified its feasibility and usability as well as its limitation and restrictions.

Its prototype has been carefully designed to meet full functionality, network transparency, interoperability, and generality of remote devices, as well as to resolve semantics differences in its network extension.

The experiments have shown that it achieves sufficient performance in a local area network for all types of devices. Its performance model has been established regarding request queuing mechanisms and network latencies.

The experiences and considerations through its development and actual usages have shown that its network extension mechanism resolves semantics differences to a degree; it devoted special care to them in its implementation and exploits parts of the semantics of the USB model.

The experiences and considerations have pointed the way to further alterations of the USB model and its implementations, since their current designs cannot fully meet the requirements of being extended over networks, especially regarding time constraints, request scheduling, bandwidth management, and request cancellation.

In addition, this dissertation has presented a device sharing system that incorporates the network extension and device management of operating systems in a consistent manner. It has shown that the network extension, which never abstracts device resources and multiplexes device access in its mechanism, can be applied to a practical application through close coordination with other device management mechanisms.

6.2 Futher Directions

The discussion about USB/IP has been performed through its prototype implementation for the Linux USB stack as well as careful investigations to other USB stacks in operating systems. However, there still remains a slight possibility of the necessity for further discussions about interoperability among different operating

systems and also suggestions for the USB model. In addition, more generalized considerations can be performed for other multipurpose peripheral interfaces through implementations of network extension via host controller virtualization. These studies will offer insight into future peripheral architecture that supports IP-network-based transport.

The proposed mechanism raises the necessity of QoS management in networks to improve its performance and stability. Although this dissertation has been primarily focused on its network extension mechanism inside end nodes, further studies about arbitration mechanisms of network transport are motivated for suitable support of fine-grained request messages of peripheral interfaces.

It also brings up new motivation of isolation mechanisms in an operating system. First, since a major part of remote device access is handled by device drivers, it is desirable that driver I/O paths are guaranteed to be independent with each other in order to avoid performance degradation by a slow path. Second, the importance of driver dependability is more attained[1], because a faulty device may be attached virtually without a visual check. Third, if the system appropriately isolates device-derived resources, it can allow more flexible device usages for non-privileged entities.

[1] An empirical study [20] showed device drivers tend to involve most bugs of operating system errors. This fact also motivates the enhancement of driver dependability.

Acknowledgments

First, I especially want to thank my family for encouragement, understanding, and warm support during long years. Without their understanding, I could not start my new academic activity in information science at Nara Institute of Science and Technology.

I would like to thank my supervisor Professor Hideki Sunahara for having accepted me as a student member of Laboratory of Internet Architecture and Systems and also for all his powerful support for my academic activity.

I would like to show my appreciation to Professor Suguru Yamaguchi who gave me warm and aggressive comments for my research. These comments were very precious for me.

I would like to thank Associate Professor Kazutoshi Fujikawa for his kind guidance for my academic activity.

I would like to thank Associate Professor Eiji Kawai who kindly taught me the nuts and bolts of researches.

My sincere gratitude also goes to all members of the laboratory for their kindhearted advices and help. I enjoyed totally five years with them.

Moreover, I also thank contributors who tried my software actively and sent me detailed feedbacks. I was very encouraged by e-mails from users and developers, who gave me the chances of exciting challenges.

Availability

The USB/IP implementation is available under the open source license GPL. The information is at `http://usbip.naist.jp/` . It has been developed with feedbacks from users and developers. In December 2005, its code was experimentally merged into the official development repository of the Linux kernel.

Actually, all USB devices work correctly and stably over IP networks in LAN environments. IP Only Server [7] by IBM Haifa Research Laboratory notes that USB/IP can be used for their proposed computer system that accesses all peripherals over IP networks. The USB/IP implementation attracts users' attention for embedded computers that have the limitation of peripheral interfaces. The application to a virtual machine is also discussed for physical device access from a guest operating system, such as Xen [6] and User Mode Linux [27]. Developers of thin client systems consider adopting the USB/IP implementation for the general support of USB devices in a terminal computer.

References

[1] IEEE Std 1394-1995. *IEEE Standard for a High Performance Serial Bus*. IEEE, August 1996.

[2] IEEE Std 1394b 2002. *IEEE Standard for a High Performance Serial Bus – Amendment 2*. IEEE, December 2002.

[3] Agere Systems, Inc., Hewlett-Packard Company, Intel Corporation, Microsoft Corporation, NEC Corporation, Koninklijke Philips Electronics N.V., and Samsung Electronics Co., Ltd. Wireless universal serial bus specification, revision 1.0, May 2005.

[4] Michael J. Accetta andRobert V. Baron andWilliam J. Bolosky andDavid B. Golub andRichard F. Rashid andAvadis Tevanian andMichael Young. Mach: A new kernel foundation for UNIX development. In *Proceedings of the USENIX Summer Conference*, pages 93–113. USENIX Association, 1986.

[5] François Armand. Give a process to your drivers. In *Proceedings of the EurOpen Autumn 1991 Conference*. EurOpen, September 1991.

[6] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, pages 164–177. ACM Press, 2003.

[7] Muli Ben-Yehuda, Oleg Goldshmidt, Elliot K. Kolodnerand Zorik Machulsky, Vadim Makhervaks, Julian Satranand Marc Segal, Leah Shalev, and Ilan Shimony. IP only server. In *Proceedings of the USENIX Annual Technical Conference*, pages 381–386. USENIX Association, 2006.

[8] Philip A. Bernstein. Middleware: A model for distributed system services. *Communications of the ACM*, 39(2):86–98, 1996.

[9] Matt Blaze, Joan Feigenbaum, Joan Ioannidis, and Angelos D. Keromytis. The KeyNote trust-management system version 2. RFC 2704, September 1999.

[10] Matt Blaze, Joan Feigenbaum, and Jack Lacy. Decentralized trust management. In *Proceedings of the 1996 IEEE Symposium on Security and Privacy*, pages 164–173. IEEE Computer Society Press, 1996.

[11] Matt Blaze, Joan Ioannidis, and Angelos D. Keromytis. DSA and RSA key and signature encoding for the KeyNote trust management system. RFC 2792, March 2000.

[12] Matt Blaze, John Ioannidis, and Angelos D. Keromytis. Trust management for IPsec. *ACM Transactions on Information and System Security*, 5(2):95–118, 2002.

[13] Matt Blaze, John Ioannidis, and Angelos D. Keromytis. Experience with the KeyNote trust management system: Applications and future directions. In *Trust Management: First International Conference, iTrust 2003*, Lecture Notes in Computer Science, pages 284–300. Springer Berlin, 2003.

[14] Bluetooth SIG, Inc. Specification of the Bluetooth system version 2.0 + EDR, November 2004.

[15] Mark Carson and Darrin Santay. NIST Net: A Linux-based network emulation tool. *ACM SIGCOMM Computer Communication Review*, 33(3):111–126, 2003.

[16] Jeffrey D. Case, Mark Fedor, Martin Lee Schoffstall, and James R. Davin. Simple network management protocol (SNMP). RFC 1157, May 1990.

[17] David Cheriton. The V distributed system. *Communications of the ACM*, 31(3):314–333, 1988.

[18] Stuart Cheshire and Marc Krochmal. DNS-based service discovery. Internet Draft, August 2006.

[19] Stuart Cheshire and Marc Krochmal. Multicast DNS. Internet Draft, August 2006.

[20] Andy Chou, Junfeng Yang, Benjamin Chelf, Seth Hallem, and Dawson Engler. An empirical study of operating systems errors. In *Proceedings of*

the Eighteenth ACM Symposium on Operating Systems Principles, pages 73–88. ACM Press, 2001.

[21] Yang-Hua Chu, Joan Feigenbaum, Brian LaMacchia, Paul Resnick, and Martin Strauss. REFEREE: Trust management for Web applications. *Computer Networks and ISDN Systems*, 29(8–13):953–964, 1997.

[22] Cluster File Systems, Inc. Lustre: A scalable, high-performance file system, November 2002.

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

[24] Compaq Computer Corporation, Hewlett-Packard Company, Intel Corporation, Lucent Technologies Inc., Microsoft Corporation, NEC Corporation, and Koninklijke Philips Electronics N.V. Universal serial bus specification, revision 2.0, April 2000.

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

[26] Tim Dierks and Eric Rescorla. The transport layer security (TLS) protocol version 1.1. RFC 4346, April 2006.

[27] Jeff Dike. A user-mode port of the Linux kernel. In *Proceedings of the 4th Annual Linux Showcase and Conference*, pages 63–73. USENIX Association, October 2000.

[28] ECHONET CONSORTIUM. The ECHONET specification, version 2.11, April 2002.

[29] Carl M. Ellison, Bill Frantz, Butler Lampson, Ron Rivest, Brian Thomas, and Tatu Ylonen. SPKI certificate theory. RFC 2693, September 1999.

[30] Mark Fasheh. OCFS2: The Oracle clustered file system, version 2. In *Proceedings of the Linux Symposium*, pages 289–302. Linux Symposium, July 2006.

[31] Roy T. Fielding, James Gettys, Jeffrey C. Mogul, Henrik Frystyk, Larry Masinter, Paul J. Leach, and Tim Berners-Lee. Hypertext transfer protocol – HTTP/1.1. RFC 2616, June 1999.

[32] Gregory G. Finn. An integration of network communication with workstation architecture. *ACM SIGCOMM Computer Communication Review*, 21(5):18–29, 1991.

[33] Alessandro Forin, David Golub, and Brian Bershad. An I/O system for Mach 3.0. In *Proceedings of the USENIX MACH Symposium*, pages 163–176. USENIX Association, November 1991.

[34] UPnP Forum. UPnP device architecture 1.0, version 1.0.1, December 2003.

[35] Keir Fraser, Steven Hand, Rolf Neugebauer, Ian Pratt, Andrew Warfield, and Mark Williamson. Safe hardware access with the Xen virtual machine monitor. In *The first Workshop on Operating System and Architectural Support for the on demand IT InfraStructure*, October 2004.

[36] Abdulbaset Gaddah and Thomas Kunz. A survey of middleware paradigms for mobile computing. Technical Report SCE-03-16, Department of Systems and Computing Engineering, Carleton University, July 2003.

[37] Shantanu Goel and Dan Duchamp. Linux device driver emulation in Mach. In *Proceedings of the USENIX Annual Technical Conference*, pages 65–74. USENIX Association, 1996.

[38] Object Management Group. Common object request broker architecture: Core specification, version 3.0.3, March 2004.

[39] Carl A. Gunter and Trevor Jim. Policy-directed certificate retrieval. *Software - Practice and Experience*, 30(15):1609–1640, 2000.

[40] HAVi, Inc. The HAVi specification, version 1.1, May 2001.

[41] Paul J.M. Havinga. *Mobile Multimedia Systems*. PhD thesis, University of Twente, February 2000.

[42] Mark Hayter and Derek McAuley. The desk area network. *ACM SIGOPS Operating System Review*, 25(4):14–21, October 1991.

[43] Jorrit N. Herder, Herbert Bos, Ben Gras, Philip Homburg, and Andrew S. Tanenbaum. MINIX 3: a highly reliable, self-repairing operating system. *ACM SIGOPS Operating Systems Review*, 40(3):80–89, 2006.

[44] Dan Hildebrand. An architectural overview of QNX. In *Proceedings of the Workshop on Micro-kernels and Other Kernel Architectures*, pages 113–126. USENIX Association, April 1992.

[45] 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*, pages 47–60. USENIX Association, April 2005.

[46] Takahiro Hirofuchi, Eiji Kawai, Kazutoshi Fujikawa, and Hideki Sunahara. USB/IP: A transparent device sharing technology over IP network. *IPSJ Transactions on Advanced Computing System*, 46(SIG 12 (ACS 11)):349–361, August 2005.

[47] 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*. IEEE Computer Society Press, March 1998.

[48] John H. Howard, Michael L. Kazar, Sherri G. Meneesand David A. Nichols, M. Satyanarayananand Robert N. Sidebotham, and Michael J. West. Scale and performance in a distributed file system. *ACM Transactions on Computer Systems*, 6(1):51–81, 1988.

[49] Chih-Yuan Huang, Tei-Wei Kuo, and Ai-Chun Pang. QoS support for USB 2.0 periodic and sporadic device requests. In *Proceedings of the 25th IEEE International Real-Time Systems Symposium*, pages 395–404. IEEE Computer Society Press, 2004.

[50] Leo J. McLaughlin III. Line printer daemon protocol. RFC 1179, August 1990.

[51] Masaaki Inami. A wired extension method for Bluetooth networks. Master's thesis, Japan Advanced Institute of Science and Technology, 2004. In Japanese.

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

[53] Intel, Hewlett-Packard, NEC, Dell. IPMI - intelligent platform management interface specification second generation v2.0, February 2004.

[54] Sotiris Ioannidis, Angelos D. Keromytis, Steve M. Bellovin, and Jonathan M. Smith. Implementing a distributed firewall. In *Proceedings of the 7th ACM Conference on Computer and Communications Security*, pages 190–199. ACM Press, 2000.

[55] Trevor Jim. SD3: A trust management system with certified evaluation. In *2001 IEEE Symposium on Security and Privacy (SP 2001)*, pages 106–115. IEEE Computer Society Press, May 2001.

[56] Stephen Kent and Karen Seo. Security architecture for the Internet protocol. RFC 4301, December 2005.

[57] Yousef Y. A. Khalidi, José M. Bernabéu-Aubán, Vlada Matena, Ken Shirriff, and Moti Thadani. Solaris MC: A multi computer OS. In *Proceedings of the USENIX Annual Technical Conference*, pages 191–204. USENIX Association, 1996.

[58] Kenji Kikuchi. IEEE1394-IEEE802.3 transparent bridges for home network. Master's thesis, Japan Advanced Institute of Science and Technology, 2002. In Japanese.

[59] Sunil Kittur, François Armand, Douglas Steel, and Jim Lipkis. Fault tolerance in a distributed CHORUS/MiX system. In *Proceedings of the USENIX Annual Technical Conference*, pages 219–228. USENIX Association, 1996.

[60] Steve R. Kleiman. Vnodes: An architecture for multiple file system types in Sun UNIX. In *Proceedings of the USENIX Summer Conference*, pages 238–247. USENIX Association, June 1986.

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

[62] Nancy P. Kronenberg, Henry M. Levy, and William D. Strecker. VAXclusters: a closely-coupled distributed system. *ACM Transactions on Computer Systems*, 4(2):130–146, 1986.

[63] Carl Kugler, Harry. Lewis, and Tom Hastings. Internet printing protocol (IPP): Requirements for job, printer, and device administrative operations. RFC 3239, February 2002.

[64] Joshua LeVasseur, Volkmar Uhlig, Jan Stoess, and Stefan Götz. Unmodified device driver reuse and improved system dependability via virtual machines. In *Proceedings of the Sixth Symposium on Operating System Design and Implementation*, pages 17–30. USENIX Association, December 2004.

[65] Eliezer Levy and Abraham Silberschatz. Distributed file systems: concepts and examples. *ACM Computer Survey*, 22(4):321–374, 1990.

[66] Peter Loscocco and Stephen Smalley. Integrating flexible support for security policies into the linux operating system. In *Proceedings of the FREENIX Track: 2001 USENIX Annual Technical Conference*, June 2001.

[67] Vlada Matena, Yousef A. Khalidi, and Ken Shirriff. Solaris MC file system framework. Technical Report TR-96-57, Sun Microsystems Laboratories, October 1996.

[68] Alex McKenzie. Telnet protocol specifications. RFC 495, May 1973.

[69] 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, December 1998.

[70] Sun Microsystems. Jini architecture specification, version 1.2, December 2001.

[71] Stefan Miltchev, Vassilis Prevelakis, Sotiris Ioannidis, John Ioannidis, Angelos D. Keromytis, and Jonathan M. Smith. Secure and flexible global file sharing. In *Proceedings of the FREENIX Track: 2003 USENIX Annual Technical Conference*, pages 165–178, 2003.

[72] James G. Mitchell, Jonathan Gibbons, Graham Hamilton, Peter B. Kessler, Yousef Y. A. Khalidi, Panos Kougiouris, Peter Madany, Michael N. Nelson, Michael L. Powell, and Sanjay R. Radia. An overview of the Spring system. In *Proceedings of COMPCON*, pages 122–131. IEEE Computer Society Press, February 1994.

[73] John Nagle. Congestion control in IP/TCP internetworks. RFC 896, January 1984.

[74] VITA Standards Organization. Myrinet-on-VME protocol specification, draft 1.1, August 1998.

[75] John K. Ousterhout, Andrew R. Cherenson, Frederick Douglis, Michael N. Nelson, and Brent B. Welch. The Sprite network operating system. *Computer*, 21(2):23–36, 1988.

[76] Rob Pike, Dave Presotto, Sean Dorward, Bob Flandrena, Ken Thompson, Howard Trickey, and Phil Winterbottom. Plan 9 from Bell Labs. *Computing Systems*, 8(3):221–254, 1995.

[77] Jon Postel and Joyce Reynolds. File transfer protocol. RFC 959, October 1985.

[78] Ian Alexander Pratt. *The User-Safe Device I/O Architecture*. PhD thesis, King's College University of Cambridge, 1997.

[79] Reconfigurable Ubiquitous Networked Embedded Systems. Survey of middleware for networked embedded systems. Technical Report IST-004536-RUNES, University College London, July 2005.

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

[81] Andrew P. Rifkin, Michael P. Forbes, Richard L. Hamilton, Michael Sabrio, Suryakanta Shah, and Kang Yueh. RFS architectural overview. In *USENIX Summer Conference Proceedings*, pages 248–259. USENIX Association, June 1989.

[82] Marc Rozier, Vadim Abrossimov, Francois Armand, Ivan Boule, Michel Gien, Marc Guillemont, Frederic Herrmann, Claude Kaiser, Sylvain Langlois, Pierre Leonard, and Will Neuhauser. Overview of the Chorus distributed operating system. In *Proceedings of the Workshop on Micro-kernels and Other Kernel Architectures*, pages 39–69. USENIX Association, April 1992.

[83] S. Masoud Sadjadi and Philip K. McKinley. A survey of adaptive middleware. Technical Report MSU-CSE-03-35, Department of Computer Science and Engineering, Michigan State University, December 2003.

[84] Farshad A. Samimi and Philip K. McKinley. A survey of kernel-middleware interaction in support of cross-layer adaptation in mobile computing. Technical Report MSU-CSE-04-44, Department of Computer Science and Engineering, Michigan State University, October 2004.

[85] Julian Satran, Kalman Meth, Costa Sapuntzakis, Mallikarjun Chadalapaka, and Efri Zeidner. Internet small computer systems interface (iSCSI). RFC 3720, April 2004.

[86] Mahadev Satyanarayanan. Scalable, secure, and highly available distributed file access. *Computer*, 23(5):9–18, 20–21, 1990.

[87] Robert W. Scheifler and Jim Gettys. The X window system. *ACM Transactions on Graphics*, 5(2):79–109, 1986.

[88] I2O SIG. Intelligent I/O (I2O) architecture specification (draft version 1.5d), March 1997.

[89] Steven R. Soltis, Grant M. Erickson, Kenneth W. Preslan, Matthew T. O'Keefe, and Thomas M. Ruwart. The global file system: A file system for shared disk storage, November 1997.

[90] Raj Srinivasan. RPC: Remote procedure call protocol specification version 2. RFC 1831, August 1995.

[91] Jeremy Sugerman, Ganesh Venkitachalam, and Beng-Hong Lim. Virtualizing I/O devices on VMware workstation's hosted virtual machine monitor.

In *Proceedings of the USENIX Annual Technical Conference*, pages 1–14. USENIX Association, 2001.

[92] Sun Microsystems. NFS: Network file system protocol specification. RFC 1094, March 1989.

[93] Michael M. Swift, Brian N. Bershad, and Henry M. Levy. Improving the reliability of commodity operating systems. *ACM Transactions on Computer Systems*, 23(1):77–110, 2005.

[94] Michael M. Swift, Steven Martin, Henry M. Levy, and Susan J. Eggers. Nooks: an architecture for reliable device drivers. In *Proceedings of the 10th workshop on ACM SIGOPS European workshop*, pages 102–107. ACM Press, 2002.

[95] T10/1161D. *Information Technology - SCSI Architecture Model - 3 (SAM-3)*. ANSI/INCITS, September 2004.

[96] David L. Tennenhouse, Joel Adam, David Carver, Henry Houh, Michael Ismert, Christopher Lindblad, Bill Stasior, David Wetherall, David Bacher, and Teresa Chang. The ViewStation: a software-intensive approach to media processing and distribution. *Multimedia Systems*, 3(3):104–115, 1995.

[97] The OpenSSI Clustering Project. OpenSSI (single system image) clusters for Linux. `http://www.openssi.org/`.

[98] Vita Nouva Holdings Limited. Inferno. `http://www.vitanuova.com/inferno/index.html`.

[99] Bruce Walker, Gerald Popek, Robert English, Charles Kline, and Greg Thiel. The LOCUS distributed operating system. In *Proceedings of the Ninth ACM Symposium on Operating Systems Principles*, pages 49–70. ACM Press, 1983.

[100] Brent Welch. A comparison of the Vnode and Sprite file system architectures. In *Proceedings of the USENIX File System Workshop*, pages 29–44, May 1992.

[101] Chris Wright, Crispin Cowan, Stephen Smalley, James Morris, and Greg Kroah-Hartman. Linux security modules: General security support for the linux kernel. In *Proceedings of the 11th USENIX Security Symposium*, pages 17–31. USENIX Association, 2002.

[102] Tatu Ylonen and Chris Lonvick. The secure shell (SSH) protocol architecture. RFC 4251, January 2006.

Appendix

USB/IP Protocol Data Unit

The protocol data unit format in usbip-0.1.4 is quoted from code. First, a basic header is defined. It is common to all types of PDU.

```
/*
 * USB/IP request headers.
 * At now, we define 4 request types:
 *
 *  - CMD_SUBMIT transfers a USB request.
 *  - RET_RETURN transfers a result of a USB request.
 *  - CMD_UNLINK transfers an unlink request of a pending USB request.
 *  - RET_UNLINK transfers a result of an unlink request.
 *
 */

/*
 * A basic header followed by other additional headers.
 */
struct usbip_header_basic {
#define USBIP_CMD_SUBMIT        0x0001
#define USBIP_CMD_UNLINK        0x0002
#define USBIP_RET_SUBMIT        0x0003
#define USBIP_RET_UNLINK        0x0004
        __u32 command;

        __u32 busnum;
        __u32 devnum;
        __u32 seqnum; /* sequential number which identifies URBs */
        __u32 pipe;
} __attribute__ ((packed));
```

`usbip_header_basic` includes, `command` (PDU type), `busnum` (destination bus), `devnum` (destination device), `seqnum` (sequence number), and `pipe` (I/O type).

A `USBIP_CMD_SUBMIT` PDU transfers a USB request to a remote Stub driver. Its result is returned by `USBIP_RET_SUBMIT`. A `USBIP_CMD_UNLINK` PDU transfers a unlink request of a pending request. When receives it, the Stub driver cancels the request as soon as possible. Its result is returned by `USBIP_RET_UNLINK`. This cancel scheme is described later in detail.

```
/*
 * An additional header for a CMD_SUBMIT request.
 */
struct usbip_header_cmd_submit {
        __u32 transfer_flags;
        __s32 transfer_buffer_length;
        __s32 bandwidth;
        __s32 start_frame;
        __s32 number_of_packets;
        __s32 interval;
        unsigned char setup[8]; /* CTRL only */
}__attribute__ ((packed));

/*
 * An additional header for a RET_SUBMIT request.
 */
struct usbip_header_ret_submit {
        __s32 status;
        __s32 actual_length; /* returned data length */
        __s32 bandwidth;
        __s32 start_frame; /* ISO and INT */
        __s32 number_of_packets;  /* ISO only */
        __s32 error_count; /* ISO only */
}__attribute__ ((packed));


/*
 * An additional header for a CMD_UNLINK request.
 */
struct usbip_header_cmd_unlink {
        __u32 seqnum; /* URB's seqnum which will be unlinked */
}__attribute__ ((packed));

/*
 * An additional header for a RET_UNLINK request.
 */
struct usbip_header_ret_unlink {
        __s32 status;
}__attribute__ ((packed));
```

`usbip_header_cmd_submit` shows an I/O request and precedes output data if available. `transfer_buffer_length` shows its I/O data size.
`usbip_header_ret_submit` shows a result of the I/O request and precedes input data if available. The actual data size performed in the device is returned by `actual_length`.

`bandwidth` and `start_frame` are reserved entries. In the Linux driver stack, the former prevents the overcommit of requests and the later offers I/O source

synchronization. Currently, the USB/IP implementation does not handle them specially (See Section 3.3.4).

```
/* the same as usb_iso_packet_descriptor but packed for PDU */
struct usbip_iso_packet_descriptor {
        __u32 offset;
        __u32 length;            /* expected length */
        __u32 actual_length;
        __u32 status;
}__attribute__ ((packed));
```

`usbip_iso_packet_descriptor` shows additional information for an Isochronous request. `offset` and `length` split the following I/O data into more fine-grained data in a constant interval.

```
/*
 * All usbip requests use a common header to keep code simple.
 */
struct usbip_header {
        struct usbip_header_basic base;

        union {
                struct usbip_header_cmd_submit  cmd_submit;
                struct usbip_header_ret_submit  ret_submit;
                struct usbip_header_cmd_unlink  cmd_unlink;
                struct usbip_header_ret_unlink  ret_unlink;
        } u;
}__attribute__ ((packed));
```

All PDU use a common header to keep code simple in the prototype.

These headers are based on the Linux driver stack. Basically, driver stacks in operating systems are quite similar, though different in detail. The protocol detail will be redesigned developing compatible implementations in other operating systems.

USB/IP Request Cancel

The cancel scheme of an I/O request is quoted from prototype code.

```
/*
 * vhci_rx gives back the urb after receiving the reply of the urb.
 * If an unlink pdu is sent or not, vhci_rx receives a normal
 * return pdu and gives back its urb.
 * For the driver unlinking the urb, the content of the urb is not
 * important, but the calling to its completion handler is important;
 * the completion of unlinking is notified by the completion handler.
 *
 * CLIENT SIDE
 *
 * - When vhci_hcd receives RET_SUBMIT,
 *
 *  - case 1a). the urb of the pdu is not unlinking.
 *    - normal case
 *    => just give back the urb
 *
 *  - case 1b). the urb of the pdu is unlinking.
 *    - usbip.ko will return a reply of the unlinking request.
 *    => give back the urb now and go to case 2b).
 *
 * - When vhci_hcd receives RET_UNLINK,
 *
 *  - case 2a). a submit request is still pending in vhci_hcd.
 *    - urb was really pending in usbip.ko
 *      and urb_unlink_urb() was completed there.
 *    => free a pending submit request
 *    => notify unlink completeness by giving back the urb
 *
 *  - case 2b). a submit request is *not* pending in vhci_hcd.
 *    - urb was already given back to the core driver.
 *    => do not give back the urb
 *
 * SERVER SIDE
 *
 * - When usbip receives CMD_UNLINK,
 *
 *  - case 3a). the urb of the unlink request is now in submission.
 *    => do usb_unlink_urb().
 *    => after the unlink is completed, send RET_UNLINK.
 *
 *  - case 3b). the urb of the unlink request is not in submission.
 *    - may be already completed or never be received
 *    => send RET_UNLINK
 */
```

USB/IP Usage Example

The following examples are quoted from README in usbip-0.1.4.

```
-----------------------------------------------------------------
- SERVER SIDE (physically attach your USB devices to this host) -
-----------------------------------------------------------------

   trois:# insmod (somewhere)/usbip.ko
   trois:# usbipd -D

In another terminal, let's look up what usb devices are physically attached to
this host.  We can see a usb storage device of busid 3-3.2 is now bound to
usb-storage driver. To export this device, we first mark the device as
"exportable"; the device is bound to usbip driver. Please remember you can not
export a usb hub.

   trois:# bind_driver --list
   List USB devices
   - busid 3-3.2 (04bb:0206)
       3-3.2:1.0 -> usb-storage

   - busid 3-3.1 (08bb:2702)
       3-3.1:1.0 -> snd-usb-audio
       3-3.1:1.1 -> snd-usb-audio

   - busid 3-3 (0409:0058)
       3-3:1.0 -> hub

   - busid 3-2 (0711:0902)
       3-2:1.0 -> none

   - busid 1-1 (05a9:a511)
       1-1:1.0 -> ov511

   - busid 4-1 (046d:08b2)
       4-1:1.0 -> none
       4-1:1.1 -> none
       4-1:1.2 -> none

   - busid 5-2 (058f:9254)
       5-2:1.0 -> hub

Mark the device of busid 3-3.2 as exportable.

   trois:# bind_driver --usbip 3-3.2
   ** (process:24621): DEBUG:  3-3.2:1.0   -> none
   ** (process:24621): DEBUG: write "add 3-3.2" to /sys/bus/usb/drivers/usbip/match_busid
   ** Message: bind 3-3.2 to usbip, complete!

   trois:# bind_driver --list
   List USB devices
   - busid 3-3.2 (04bb:0206)
           3-3.2:1.0 -> usbip
   (snip)

Iterate the above operation for other devices if you like.
```

```
---------------
- CLIENT SIDE -
---------------
First, let's list available remote devices which are marked as exportable in
the server host.

   deux:# insmod (somewhere)/vhci_hcd.ko

   deux:# usbip --list 10.0.0.3
   - 10.0.0.3
          1-1: Prolific Technology, Inc. : unknown product (067b:3507)
             : /sys/devices/pci0000:00/0000:00:1f.2/usb1/1-1
             : (Defined at Interface level) / unknown subclass / unknown protocol (00/00/00)
             :  0 - Mass Storage / SCSI / Bulk (Zip) (08/06/50)

      1-2.2.1: Apple Computer, Inc. : unknown product (05ac:0203)
             : /sys/devices/pci0000:00/0000:00:1f.2/usb1/1-2/1-2.2/1-2.2.1
             : (Defined at Interface level) / unknown subclass / unknown protocol (00/00/00)
             :  0 - Human Interface Devices / Boot Interface Subclass / Keyboard (03/01/01)

      1-2.2.3: OmniVision Technologies, Inc. : OV511+ WebCam (05a9:a511)
             : /sys/devices/pci0000:00/0000:00:1f.2/usb1/1-2/1-2.2/1-2.2.3
             : (Defined at Interface level) / unknown subclass / unknown protocol (00/00/00)
             :  0 - Vendor Specific Class / unknown subclass / unknown protocol (ff/00/00)

          3-1: Logitech, Inc. : QuickCam Pro 4000 (046d:08b2)
             : /sys/devices/pci0000:00/0000:00:1e.0/0000:02:0a.0/usb3/3-1
             : (Defined at Interface level) / unknown subclass / unknown protocol (00/00/00)
             :  0 - Data / unknown subclass / unknown protocol (0a/ff/00)
             :  1 - Audio / Control Device / unknown protocol (01/01/00)
             :  2 - Audio / Streaming / unknown protocol (01/02/00)

          4-1: Logitech, Inc. : QuickCam Express (046d:0870)
             : /sys/devices/pci0000:00/0000:00:1e.0/0000:02:0a.1/usb4/4-1
             : Vendor Specific Class / Vendor Specific Subclass / Vendor Specific Protocol (ff/ff/ff)
             :  0 - Vendor Specific Class / Vendor Specific Subclass / Vendor Specific Protocol (ff/ff/ff)

          4-2: Texas Instruments Japan : unknown product (08bb:2702)
             : /sys/devices/pci0000:00/0000:00:1e.0/0000:02:0a.1/usb4/4-2
             : (Defined at Interface level) / unknown subclass / unknown protocol (00/00/00)
             :  0 - Audio / Control Device / unknown protocol (01/01/00)
             :  1 - Audio / Streaming / unknown protocol (01/02/00)

Attach a remote usb device!

   deux:# usbip --attach 10.0.0.3 1-1
   port 0 attached

Show what devices are attached to this client.

   deux:# usbip --port
   Port 00: <Port in Use> at Full Speed(12Mbps)
          Prolific Technology, Inc. : unknown product (067b:3507)
          6-1 -> usbip://10.0.0.3:3240/1-1  (remote bus/dev 001/004)
          6-1:1.0 used by usb-storage
                         /sys/class/scsi_device/0:0:0:0/device
                         /sys/class/scsi_host/host0/device
                         /sys/block/sda/device

Detach the imported device.

   deux:# usbip --detach 0
   port 0 detached
```

Assertion Parameters

Table A.1 is assertion parameters defined for the prototype sharing system in Section 4.4.2.

Table A.1: Assertion Parameters

uid	subject UID
gid	subject GID
host	subject host name
af	address family
ip	subject IP address
peerkey	subject public key
devhost	origin host
bus	bus name
busid	device identifier on the bus
time	attachment time
idVendor	USB vendor ID
idProduct	USB product ID
bDeviceClass	USB class ID
bDeviceSubClass	USB subclass ID
bDeviceProtocol	USB protocol ID
manufacturer	USB manufacturer name
product	USB product name
serial	USB serial number (if available)
bInterfaceClassN	USB the Nth interface class ID
bInterfaceSubClassN	USB the Nth interface subclass ID
bInterfaceProtocolN	USB the Nth interface protocol ID
mode	LocalDev, ExportDev, RemoteDev
lease_time	elapsed time from the last acquisition

Publications

Journal

- Takahiro Hirofuchi, Eiji Kawai, Kazutoshi Fujikawa, and Hideki Sunahara. USB/IP: A Transparent Device Sharing Technology over IP Network. *IPSJ Transactions on Advanced Computing System*, 46(SIG 12 (ACS 11)):349–361, August 2005.

International Conference

- 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*, pages 47–60. USENIX Association, April 2005.

Technical Report

- Takahiro Hirofuchi, Eiji Kawai, Yutaka Nakamura, Kazutoshi Fujikawa, and Hideki Sunahara. The Proposal of a Device Control Framework with USB over IP for Realizing IP Device Space. In *IPSJ SIG Technical Reports*, 2003-UBI-2, pages 117–112. Information Processing Society of Japan, November 2003. In Japanese.

- Takahiro Hirofuchi, Yutaka Nakamura, Eiji Kawai, Kazutoshi Fujikawa, and Hideki Sunahara. A Device Access Method over Network by Extending USB Driver Stack of Linux. In *IPSJ SIG Technical Reports*, 2003-OS-93, pages 41–48. Information Processing Society of Japan, March 2003. In Japanese.

Award

- USB/IP - A Peripheral Bus Extension for Device Sharing over IP Network. FREENIX Track Best Paper Award: 2005 USENIX Annual Technical Conference. USENIX Association, April 2005.

- USB/IP: A Transparent Device Sharing Technology over IP Network. IPSJ Digital Courier Funai Young Researcher Award. Information Processing Society of Japan, April 2006.