Design and Implementation of Peripheral Sharing Mechanism on Pervasive Computing with Heterogeneous Environment

Wonhong Kwon[1], Han Wook Cho[1], Yong Ho Song[1]

[1] College of Information and Communications, Hanyang University,
Seoul, Korea
{whkwon, hwcho, yhsong}@enc.hanyang.ac.kr

Abstract. As pervasive computing permeate into user's lives, many embedded devices based on Linux exist around the users. In this circumstance, the heterogeneousness of operating systems causes incompatibility problems in sharing peripherals since the users and the devices have a different operating system. In this paper, we propose a USB Cross-platform Extension to share peripherals in a heterogeneous environment via a TCP/IP network. Using our approach, the users can access remote peripherals with different operating systems as if they were attached to a local computer. According to our evaluation results, our approach has some overhead, but sufficient performance for practical usage.

Keywords: Peripheral sharing, Heterogeneous operating system environment, Pervasive computing

1 Introduction

Recent advances in computing technology have enabled ubiquitous computing environments to permeate users' lives rapidly. In such an environment, a number of embedded devices such as PDA, MP3 player, or cell phone, exist around the users. These devices usually use Linux as their operating system because of its characteristics such as reconfigurability, flexibility, lightweight size, and cheap price. However those who use these devices are more familiar with Microsoft Windows rather than Linux.

The heterogeneousness of operating systems causes incompatibility problems in sharing peripherals between users and embedded devices. For instance, a user who uses Windows may want to access an external storage device in a PDA based on Linux. Also, the user may want to use a speaker connected to a MP3 player or a DVD-RW in home appliances. In these scenarios, if the user's computer and the embedded devices use a same operating system, they can easily share peripherals. If the devices do not use the same operating system, however, they need a way of sharing peripherals in a heterogeneous environment.

Many peripheral sharing mechanisms have been proposed for heterogeneous environments. For instance, SAMBA [1] is a well-known file and print service protocol between Windows clients and non-Windows servers via a TCP/IP network. Using the SAMBA, a user can access files or printers located at remote machine with a non-Windows operating system. However, traditional approaches including SAMBA depend on specific peripherals. Therefore, these approaches are inappropriate for the recent variety of sophisticated peripherals in pervasive computing. In these circumstances, there is a great need of peripheral-independent sharing mechanism in heterogeneous operating system environment.

In this paper, we propose USB Cross-platform Extension (UCE) to share peripherals in a heterogeneous environment via a TCP/IP network. We assume that all computers and embedded devices are connected together via the network and they have heterogeneous operating systems. In this environment, our approach enables users to access remote USB peripherals with different operating systems as if they were attached to a local machine. USB was chosen because it supports almost all devices including storage, keyboard, speaker, and printer. This characteristic gives peripheral-independence to our approach.

The rest of this paper is organized as follows. Section 2 introduces related work. Section 3 describes the general concepts of Windows USB System and Section 4 explains the USB Cross-platform Extension. Section 5 explains our evaluation results. We conclude this paper in Section 6.

2 Related Works

USB Cross-platform Extension is based on USB over IP technology to provide sharing mechanism in a heterogeneous environment. Takahiro et al. proposed USB/IP [2][3] is a peripheral bus extension over a TCP/IP network in a homogeneous environment. Virtual Host Controller Interface (VHCI) Driver and Stub Driver were added to Linux to extend the peripheral bus via the network. VHCI located at a client-side which required to access remote devices, is responsible for processing enqueued USB Request Block (URB) like legacy USB Host Controller in Linux. When VHCI receives the URBs, they are converted into USB/IP packets and sent to a remote machine. Stub Driver is a new USB Per-Device Driver and located at the remote machine. It decodes incoming USB/IP packets, extracts the URBs and submits them to devices. These new drivers enable users to share a large range of devices over the network without any modification in existing components. This mechanism supports all USB transfer features such as bulk, interrupt, control, and isochronous mode and its I/O performance is sufficient for practical usage.

We extend this mechanism to heterogeneous environment. In the context of our approach, the operating system of a client is Windows instead of Linux in USB/IP. This requires that the VHCI is migrated to Windows for processing URBs and sending them to the remote machine. Therefore, we create UCE Bus in Windows as a new USB bus architecture to extend peripheral bus to other operating systems.

In the next section, we use Windows USB System to explain how UCE is designed and implemented.

3 Windows USB System

Windows supports USB in various ways such as system-provided USB Function Driver (usbhid.sys, usbstor.sys), USB Bus Driver (usbhub.sys), USB Host Controller Driver (usbohci.sys, usbuhcd.sys, usbehci.sys), and USB common library (usbd.sys) to provide convenience for users and developers. Fig. 1 shows the architecture of these drivers in Windows USB System.

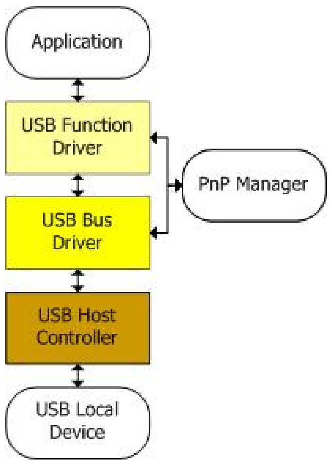

Fig. 1. Windows USB System Architecture

USB Function Driver has the information about specific hardware and provides controlling methods of the hardware to applications. When a device is attached, Windows dynamically loads the appropriate USB Function Driver with the information of the device. While this driver is active, it provides an interface to applications supporting I/O requests and translates these requests to a USB-specific format called URB.

USB Bus Driver manages USB devices that are currently attached and supports self-identifying of newly attached devices. Self-identifying, one of the important features of USB, is where configuring a device automatically occurs without additional steps taken by the user. This driver cooperates its work with PnP (Plug and Play) Manager because managing and self-identifying require various PnP operations. USB Host Controller Driver is responsible for the notification of device attachment. USB Root Hub on this controller sends a message to the USB Bus Driver when a device is arrived.

Fig. 2 shows that how these drivers work together. When a USB device is attached to the host, USB Host Controller Driver senses this attachment and sends a message to USB Bus Driver called Hot-plug Notification in MSDN [4]. If that happens, USB Bus Driver allocates resources for the device and sends a message indicating that a relation of attached devices is changed to PnP Manager using a Windows kernel API called IoInvalidateDeviceRelations(). Because a change of relation is generated when a device is attached or removed, PnP Manager needs to confirm the cause of a change.

Therefore, PnP Manager requests a list of devices to USB Bus Driver to identify the changes. USB Bus Driver responds to this message with an updated device list, and PnP Manager recognizes the cause of a change and performs various PnP operations as required. Then the PnP Manager loads the proper USB Function Driver based on previous PnP operations, allocates resources for the device, and starts the device. Even if the USB Function Driver has been loaded, it does not know device-specific information but only general information about the device. For instance, usbhid.sys is system-provided USB Function Driver for a USB keyboard. It has routines for general USB keyboard processes but does not know that how many keys are in the keyboard or which information is displayed on the LED. USB specification [5] provides USB descriptors to USB Function Driver to support this device-specific information. USB Function Driver is self-configured with USB descriptors and performs additional device-specific operations as required. After all of the processes have been completed, the architecture of these drivers for supporting the device is established and applications can use the device.

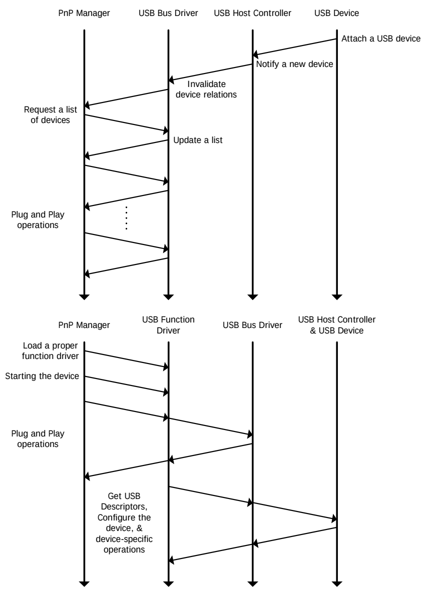

Fig. 2. Initialization process of a local device

4 USB Cross-platform Extension

USB Cross-platform Extension (UCE) is a peripheral sharing mechanism via a TCP/IP network for heterogeneous environment. In this section, we explain the architecture of UCE and describe UCE Bus which is a main component of our approach.

4.1 UCE Architecture

Using the UCE, a user can use a remote device in heterogeneous operating system as if the device were attached locally and working in the same operating system. It means that UCE provides transparency to the user by hiding a location and an operating system of the device. To achieve this, we attach the device to a local machine not physically but conceptually and we call this “virtual device.”

UCE has different device driver architecture than the Windows USB System due to the virtual device. This is because the virtual device has some extraordinary characteristics compared to the local one. Therefore, Windows USB System as we mentioned in Section 3 is not directly applicable to our approach in two ways. First, the virtual device is not a real device. A legacy system sometimes needs to get the information from a device like USB Descriptors. However, the virtual device does not have this information because it only exists conceptually. In addition, the Windows USB Bus cannot receive a notification message indicating that a new device is attached from USB Host Controller because no devices are attached to local machine physically. Second, the real location of the virtual device is not local but remote. Therefore, a remote machine must be accessed via the network but a legacy system does not support this. Because of these problems, a legacy system needs modification supporting the virtual device. Fig. 3 shows our proposed methods.

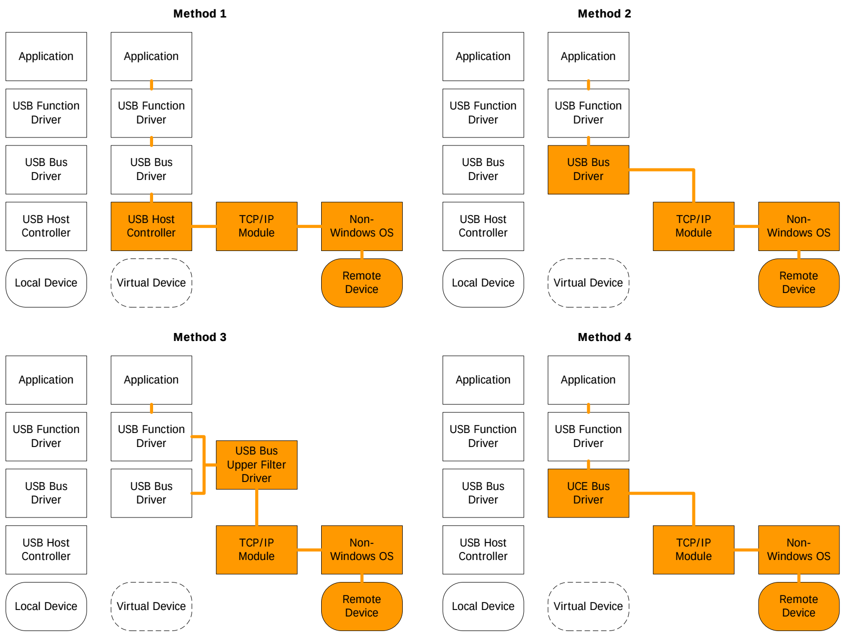

Fig. 3. Design candidates of UCE

Method 1 and 2 are quite a simple and easy to implement compared to others because these methods reuse large portions of the legacy system. For instance, UCE needs a communication module to access a remote device, so we added a TCP/IP module to the legacy system. The TCP/IP module is attached to USB Host Controller in method 1 and USB Bus Driver in method 2. This approach requires less modification to UCE. However, Microsoft does not release source codes for the USB Bus Driver and Host Controller because of their security policies. That is, we cannot modify the USB Bus Driver or Host Controller to add the TCP/IP module. In this respect, these methods are inappropriate for our approach.

Method 3 uses USB Bus Upper Filter Driver to cooperate with the TCP/IP module. Because it is located between USB Function Driver and USB Bus driver, every packet generated from USB Function Driver is arrives in it. If that happens, it has three methods for processing the enqueued packets: passing packets to a lower layer, processing and passing packets, or processing and completing packets. Among these methods, we chose the processing and completing method when USB Function Driver requests the information from a remote USB device. That is, packets are delivered to a remote machine in which a remote device resides and are completed after receiving a response from a remote machine. In other cases such as device initialization or PnP operations, we chose a passing method which is simply transfers packets to the Windows USB Bus. However, this two-way approach has a serious problem. When Windows USB Bus needs information from the device, this approach does not pass the packets to a remotely-attached device because of the lack of connection between the Windows USB Bus and the TCP/IP module.

Consequently, we proposed method 4 using UCE Bus for connecting with the TCP/IP module. Unrevealed source codes for USB Bus Driver and USB Host Controller do not need anymore because of creating a new USB bus. And we eliminate a two-way problem in terms of attaching a TCP/IP module to UCE Bus directly. However, the abandonment of lots of portion in a legacy system causes the difficulty of implementation and several other problems.

4.2 Implementation of UCE Bus

The UCE Bus is the main component of our approach. We implemented our approach based on Windows to access remote devices connected to non-Windows operating systems. Although there are many non-Windows OS, this paper focuses on Linux which is commonly used for embedded systems.

We extend the Windows USB System to UCE Bus to support virtual devices, but they are completely independent and do not interfere with each other. For instance, the existing local devices are attached to Windows USB Bus and virtual devices are attached to UCE Bus. The independence of these buses does not affect the upper layers of them such as USB Function Drivers or applications. This means that UCE Bus provides transparency to upper layers and they are still available without any changes.

UCE Bus is composed of three components for supporting transparency to upper layers and communicating with Linux. They are IRP Virtual Processing, URB Conversion, and the Transport Driver Interface.

IRP Virtual Processing. IRP (I/O Request Packet) is a data structure to communicate between Windows and kernel-mode device drivers. In order to provide transparency to upper layers, UCE Bus handles IRPs like Windows USB Bus.

However, we cannot know the behaviors of USB Bus Driver exactly because of its unrevealed source codes, so these IRPs are handled virtually. In order to make this easier, we use USB Bus Upper Filter Driver to analyze behaviors of USB Bus Driver. As we mentioned before, filter driver receives all IRPs which pass through itself. We get these IRPs using WinDBG [6], and make UCE Bus more likely to act as Windows USB Bus. For instance, an IRP named with IRP_MN_QUERY_CAPABILITIES is requested by PnP Manager to get the information of a device such as power status, approval of removing suddenly from a host without ejection process. UCE Bus receives this IRP when the device is enumerated, but before the USB Function Driver is loaded for the device. In most case, USB Bus Driver sets any relevant values in the DEVICE_CAPABILITIES structure and returns it to the PnP Manager. However, UCE Bus does not know the information related to the device because it has a virtual device. Therefore, we handle this IRP virtually based on our analysis of the results with USB Bus Upper Filter Driver. Table 1 shows IRPs handled virtually in UCE Bus.

Table 1. IRPs with IRP_MJ_PNP as a major function code

IRPs
IRP_MN_QUERY_DEVICE_RELATIONS
IRP_MN_QUERY_ID
IRP_MN_QUERY_CAPABILITIES
IRP_MN_DEVICE_TEXT
IRP_MN_QUERY_RESOURCE_REQUIREMENTS
IRP_MN_QUERY_BUS_INFORMATION
IRP_MN_RESOURCES
IRP_MN_QUERY_LEGACY_BUS_INFORMATION
IRP_MN_FILTER_RESOURCE_REQUIREMENTS

URB Conversion. USB Request Block is a packet format used by device drivers related to USB when a driver needs communication with other drivers. Windows and Linux use URB but the structure and data types are different. For instance, URB has pipe information which indicates direction of transfer (host-to-device or device-to-host), type of pipe (isochronous, interrupt, control, or bulk), and endpoint address. Both operating systems have this information but different presentation. The variable that indicates the interrupt type of pipe has a value 3 in Windows and 1 in Linux. To solve this problem, UCE Bus has URB Conversion component to convert URB for Linux Stub Driver. This conversion process is based on USB over IP technique in USB/IP.

Transport Driver Interface. UCE Bus needs to use a TCP/IP network to communicate with Linux. Therefore, we apply Transport Driver Interface (TDI) to UCE Bus. TDI is a kernel-mode network interface in Windows and it supports all transport protocol stacks and is used particularly for a TCP network in our approach.

Fig 5 shows the attachment and initialization of a virtual device. In a legacy case, Windows USB Bus notices the attachment of a device by a message from the USB Host Controller Driver. However, UCE Bus does not have a method of confirming the attachment. In this respect, UCE Bus sends a message to a remote machine to identify available remote devices. Once this process is done, the remaining processes are the same compared with Fig 2 excluding virtually processing IRP and converting URB.

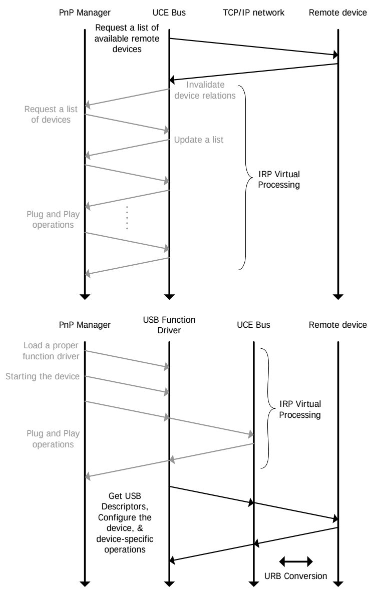

Fig. 5. Initialization process of a virtual device

UCE Bus needs to confirm the type of IRP before using these components. IRP has major and minor function codes to indicate its type. USB Function Driver sends URB to UCE Bus via IRP which has IRP_MJ_INTERNAL_DEVICE_CONTROL in a major function code. In other words, UCE Bus receives general IRP when it receives IRP without IRP_MJ_INTERNAL_DEVICE_CONTROL. Based on this information, UCE Bus activates the IRP Virtual Processing component for general IRP and the URB Conversion component for URB.

IRP Virtual Processing component receives general IRPs related to power management or PnP operation. These IRPs are completed immediately after processing because they are processed virtually in a local machine. This process is simple and does not require much time. However, URB Conversion component receives URBs and most of them need complicated works. For instance, USB Function Driver requests a read operation to an USB storage device. This operation requires serialized I/O and it has a great deal of overhead even if it is processed locally. In our situation, received URBs are not completed after processing because we transmit URBs to a remote machine in addition to the original overhead. Therefore, URBs are pending immediately when they are arrive and are completed after processing with a remote machine.

Once URB is pending, URB Function Driver can request another URB. To handle this, we maintain IRPs in pending IRP queue until they are completed. URB Conversion component checks this queue repeatedly and when IRP is found it starts a process with receiving and sending modules in TDI Client. URB Conversion component and two modules in TDI Client are made by thread for concurrency process.

5 Evaluation

In order to evaluate the performance of our approach, following environment was used as shown in Table 2.

Table 2. Evaluation environment

Client	
CPU	Intel Pentium 4 3.20GHz (dual core)
Memory	1024MB
OS	Microsoft Windows XP Professional
Server	
CPU	Mobile Intel Celeron 2.0GHz
Memory	510MB
OS	Linux 2.6.13

USB has four transfer features which are isochronous, interrupt, bulk, and control. Among these features, we evaluate interrupt transfer mode for human interface devices like mouse or keyboard. Our target device is a SAMSUNG USB SEM-DT35 keyboard. We measured response time which indicates the period of processing enqueued URB. We tested the device when it was attached to local and remote in order to evaluate our approach compared with legacy Windows USB System.

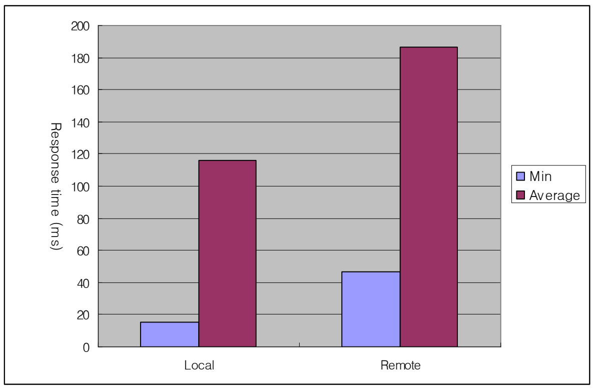

Fig. 6. Evaluation results

First, we evaluated the local device. When URB is generated, it is processed by USB Bus Driver, USB Host Controller Driver, and USB device. And it is completed in reverse direction. Response time in this evaluation includes all of these processes.

According to our measurement, the average response time was 116ms. In second evaluation, the device was attached to a remote machine with Linux. Therefore, generated URB was processed by UCE Bus, TCP/IP network, Stub Driver in Linux, and remote USB device. Average response time was 186ms. As Fig. 6 shows, response time of the remote device was 1.6 times longer than one of the local device. Although performance of UCE was lower than Windows USB System, it is sufficient for practical use.

6 Conclusion

This paper proposed USB Cross-platform Extension that is a peripheral sharing mechanism in a heterogeneous environment via a TCP/IP network. Our approach enables users to access remote USB devices as if they were attached to a local machine using the same environment.

In order to design and implement our approach, we apply UCE to Windows and exploit Linux as the target heterogeneous environment. UCE Bus is a virtual peripheral bus based on Windows and provides virtual attachment and transparency to local machine for accessing remote devices regardless of concerning their location and operating systems. According to our evaluation results, performance of remote USB devices attached to UCE Bus virtually is 1.6 times less than local one, but it is sufficient for practical use.

Acknowledgments. This research work has been supported by Nano IP/SoC Promotion Group of Seoul R&BD Program in 2006 and the Brain Korea 21 Project in 2006.

References

1. SAMBA, http://us1.samba.org/samba/
2. Takahiro Hirofuchi, Eiji Kawai, Kazutoshi Fujikawa, and Hideki Sunahara, “USB/IP – a Peripheral Bus Extension for Device Sharing over IP Network,” In the Proceedings of the FREENIX Track: USENIX Annual Technical Conference, April, 2005, pp. 47-60
3. Takahiro Hirofuchi, Eiji Kawai, Kazutoshi Fujikawa, and Hideki Sunahara, “USB/IP: A Transparent Device Sharing Technology over IP Network”, IPSJ Transactions on Advanced Computing Systems, August, 2005, pp. 349-361
4. MSDN, http://msdn2.microsoft.com/en-us/library/default.aspx
5. Universal Serial Bus Revision 2.0 specification, http://www.usb.org/developers/docs/
6. WinDBG, http://www.microsoft.com/whdc/devtools/debugging/debugstart.mspx