REMOTE USB PORTS

A Thesis

by

RAKESH ROSHAN

Submitted to the Office of Graduate and Professional Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

MASTERS OF SCIENCE

Chair of Committee,	Riccardo Bettati
Committee Members,	A. L. Narasimha Reddy
	Radu Stoleru
Head of Department,	Nancy Amato

December 2013

Major Subject: Computer Science And Engineering

ABSTRACT

Simplicity, easy to install, plug & play, high bandwidth, low latency and source of power are features of USB devices. Due to these features, many sensors and actuators are manufactured with USB interfaces for use in industries. The sensors and actuators need to be installed in fields. A computer system with USB interfaces is required to be present at the location of USB device for its working. In industry, these sensors and actuators are scattered over a large geographical area. The computers connected to them expose a large attack surface. These computers can be consolidated using virtualization and networking to reduce the attack surface. In order to consolidate computers, we need solution to extend USB port over networks so that, a USB sensor or actuator, placed in fields can be accessed by a system remotely and securely.

In this thesis, we propose a remote USB port, which is an abstraction of a USB port. In the USB core driver of the server machine, with the hub information, port status of all the ports is stored in a port status table. On the client machine a virtual host driver is created to manage proxy USB ports. When a device is inserted or removed from the USB port on the server, the client gets notified and corresponding device driver is loaded or unloaded respectively. To secure URBs, URB headers are encrypted before sending them over networks. We have implemented our solution in the Linux 3.5 kernel. We tested our solution on two machines connected over a 100 Mbps network. Various different types of USB devices were connected in the server machine and tested from the client machine. We found our solution to be device, device driver and USB protocol independent and transparent to network and device failures.

DEDICATION

I dedicate my thesis to my parents K. B. Singh and Urvashi Singh for their unconditional love, and unwavering moral and emotional support.

ACKNOWLEDGEMENTS

I would like to acknowledge my committee chair Dr. Riccardo Bettati for his continuous guidance and unwavering support throughout my thesis, without whom this thesis would not have been possible. His enthusiasm, encouragement and faith in me throughout have been extremely helpful. He was always available for my questions. He was always very generous in giving me time and help me with his vast knowledge. He always helped in solving problems with his knowledge or pointing to right source and path to solve the problem. He helped a lot in writing this thesis.

I am very thankful to my parents K. B. Singh and Urvashi Singh for their unwavering moral and emotion support. I am also very thankful to my fiancee Swarnica Pankaj for her love and support.

I would like to thank my committee members Dr. Narasimha Reddy and Dr. Radu Stoleru for their generous time and valuable suggestion whenever I needed help.

I am grateful to the TAMU writing center for their valuable suggestion while writing this thesis.

NOMENCLATURE

VHCD	Virtual Host Controller Driver
USB	Universal Serial Bus
URB	USB Resource Block
TCP	Transmission Control Protocol
IP	Internet Protocol

TABLE OF CONTENTS

LIST OF FIGURES

FIGURE Page

LIST OF TABLES

TABLE Page

1. INTRODUCTION

A number of highly visible recent attacks on factory automation and SCADA infrastructure (most prominently the Stuxnet worm [1]) have illustrated the vulnerability of such systems to security attacks. Important factors that contribute to this vulnerability are:

1. These systems are distributed over a large area, with important and vulnerable components deployed in the field and thus, they are difficult to manage and control.

2. Systems targeted by such attacks are highly complex and multilayered. They are composed of a variety of software from different vendors. These components interact and offer a particularly rich and vulnerable attack surface.

Means must be found to protect such applications by consolidating system components as much as possible and thus, reducing the attack surface in the geographic domain and by simplifying the field-deployed components.

Unfortunately, traditional consolidation techniques used in enterprise systems (for example, moving services to the cloud) cannot be applied directly, as factory automation and SCADA systems are inherently cyberphysical in nature. Field components have to communicate directly with sensors and activators, which are typically not accessible through networking capable interfaces. At least in the near future, methods must be found to extend traditional device access interfaces across inter-networks.

With very high bandwidth communication available, we can communicate with devices over networks with little delay. Gigabit Ethernet [2] supports up to 10 Gbps speed with low latency. In such high-bandwidth and low latency environments, communicating with field-deployed devices over networks will result in high performance.

As an initial study, we set out in this thesis to investigate the possibility to extend the USB device access protocol over IP. Today USB ports are ubiquitous. USB devices are easy to handle. It supports Plug and Play. The latest USB 3.0 [3] supports a data transfer rate of up to 5 Gbps with low latency. With such high data transfer rate and different traffic types, USB supports a large range of devices.

We are proposing a solution in which an individual USB port can be exported to a remote machine and remotely accessed. Whenever a device is connected to such an exported USB port, the device becomes available at the remote machine and corresponding drivers are loaded there. The local machine attached to the device (the USB server) will not have access to the functionality of the device. Instead, it forwards data and control commands between the remote machine (the USB client) and the device. The operating system functionality on the USB server can be minimal. In the application consolidation scenario, as is described above, the benefits of a scaled down functionality at the USB server host would further reduce the vulnerable attack surface offered by the large scale distributed systems. On one hand, application components of large-scale systems can be consolidated and hence, better protected. In addition, exporting of USB ports and hence, the devices connecting to those ports, would eliminate the need to deploy potentially vulnerable device drivers in the machines deployed over a large area in the field. Finally, we will show how the same framework used to export USB ports can be used to disable individual ports as well, and thus further strengthen field deployed components against attacks. The semantics of remote access to USB devices remains an intuitive one. If a USB client machine fails, the application can be started on a different machine and can be reconnected to the device. Similarly, if the device at the USB server machine fails, this is treated as a device disconnection at the client. While we limit the description to

USB bus protocol only, we envision that other bus protocols such as, PCI, HDMI etc. can also be extended over networks to remote devices.

This thesis is organized as follows: In Section 2, we will describe criteria which have been followed for a good solution. In section 3, we will discuss existing work related to computer bus extension over networks. In Section 4, a brief background of USB device model is discussed. In Section 5, we will give a full design and architecture of our solution. In Section 6, we will evaluate the system designed in Section 5, and discuss the results.

2. SOLUTION CRITERIA

USB is a very popular interface to connect a myriad of devices to computing hosts. It operates at speeds up to 5 Gbps with low latency. It supports dynamic configuration of device and four different types of traffic. In order to extend it over the internet, we have to maintain these features. In addition to correctly supporting the USB specifications, we have to keep the following criteria in mind.

2.1 USB Protocol Independent

The USB specification supports several different USB protocls each for different types of devices. USB 1.1 [4] operates at full speed (12 Mbps) and low speed (1.5 Mbps). Human interface devices like keyboards etc. operate at these speeds. USB 2.0 [4] operates at high speed (480 Mbps) and is used by high-speed devices like camera etc. USB 3.0 [3] operates at a super speed(5 Gbps) and is used for very high performance peripheral such as storage devices, displays and others. The solution should identify the device speed and behave accordingly. It should work with all USB protocols. USB supports four types of traffic: Isochronous, Interrupt, Control and Bulk. In interrupt traffic, at an interval set by driver, host fetches packet from the device. It is used for the purpose in which interrupt was used in earlier connection types like mouse. The host schedules an IN or OUT transaction at the interval. It supports small sized packets. In isochronous traffic, a bandwidth is guaranteed by the host. It is periodic in nature. Control traffic is used to set and get configuration of device. Bulk traffic is used for devices which transfer large non time-sensitive data. The bulk, control and interrupt USB Resource Blocks (URBs) are similar and there is no difference in structure. In isochronous traffic, a URB encapsulates child packets. So for isochronous traffic, the serialization and deserialization process should handle it differently from the other URBs. It should support all traffic types.

2.2 Device Agnostic

The solution should not be dependent on the devices, attached in the USB ports. On client machine, it should behave seamlessly as if the device is connected/disconnected to its local USB port. If the client machine has a proper device driver for the device which is connected to one of its remote USB port, then the device driver should be loaded and the device should be ready to use by any application on the client machine. Our solution should not impose any restriction on it. The solution should work smoothly whether a device is connected to the USB port or not. The process of connection and disconnection of a device on a remote USB port in a client machine should be such, that they are done directly on the client machine.

2.3 Device Driver Independent

Device drivers should not require any changes to use a device connecting to a remote USB port. In fact, device drivers should be totally oblivious to the fact that the device is connected to a remote USB port. It should continue to work similar to the way as it works with a device connected to a local USB port. Specifically, the fact that the device driver controls a device connected to a remote USB port must be fully transparent to the device driver except the performance transparency.

2.4 Bandwidth And Latency

The performance loss in particular, bandwidth loss and latency of exporting a port should be minimum. In particular, the bandwidth should only be limited by the network bandwidth. Similarly, the latency overhead caused by accessing a device over an exported port must be kept minimized. The protocol being used should not cause much overhead over the USB protocol overhead. The protocol information should be minimal to successfully transfer the USB packets across the network. Today with the large availability of Gigabit Ethernet, the network latency has decreased a lot. With less protocol overhead, the bandwidth of device should be maintained close to the device bandwidth.

2.5 Transparent Failure Semantics

With network in between the device and the machine, different types of failures can occur such as, machine failure, device failure and network failure. The machine with exported port may fail. The device connected to an exported port may fail. The network connecting the two machines may fail. The solution should handle them transparently. The device driver, functioning at the USB client machine should be oblivious to such failures. Remote machine failure, network failure or device failure, all such failures should be abstracted under a failure which device driver can handle. Thus same device driver can work smoothly with a remote USB port.

2.6 Security

Network is an insecure medium. We have to take care of integrity of URB transferring over networks and authentication of clients and server. An authorized user only should be allowed to connect to a USB port remotely. The data being transferred should be secured and their integrity should be maintained. The remote machine should also be ensured that the data is coming from a reliable source. The data would need to be transferred between kernels directly so, they should be secured from network attacks. At the same time we have to take care of bandwidth and latency too. Making the protocol more secure will increase the latency and hence, the bandwidth will be reduced. The protocols should be secure but, with less protocol and security overhead.

3. RELATED WORK

Over the years, a number of bus protocols have been extended over networks in a variety of ways. The aim is to make devices that are connected to such buses become available to remote machines transparently. Examples range from buses supporting system-attached storage, to USB bus and finally to High speed system buses. Some of these extensions are done over IP (iSCSI, Modbus TCP/IP, USBIP), while others simply replace the physical medium from wired to wireless (wireless USB).

3.1 iSCSI

The Small Computer System Interface (SCSI) [5] defines the physical interface and the set of commands to provide communication between peripheral devices that are connected to the computer. It is used for storage devices, scanners, CD drives and others. A SCSI device is of three types: an *initiator* device, a *target* device and a *initiator/target* device. The initiator initiates a command and the target receives initiator's command and replies with the requested I/O transfer. The initiator/target device has property of both the initiator device and the target device. For example, in a typical setting a computer is an initiator and a hard disk is a target device. SCSI over IP (iSCSI) [6] defines a standard to extend the basic SCSI across an IP network. In iSCSI the medium is the network instead of the standard SCSI connections. The SCSI commands and data are transported between a target and initiator in TCP packets. This enables a target device to be connected to a remote initiator, provided the machine with the target device and machine with the initiator are connected over a network. The iSCSI protocol is independent of device and operating system. Target and initiator can be on two machines with two different operating systems. iSCSI supports authentication but does not support encryption. In iSCSI the target machine provides the list of targets available and the initiator connects to any of them using their addresses. A device need to be remain connected for the protocol to work.

3.2 USBIP

USBIP [7] is an approach to make USB devices accessible over an IP network. A virtual bus is implemented, which abstracts all the network layer activities. In this technology, USB packets (URBs) are exchanged over a network between the device on the remote machine (the latter is called *server*) and the device driver on the client machine. A device connected to the server is bound to a special *stub device driver*. On the client machine, a virtual host controller is installed, which talks to a stub driver and passes any URB sent to it to the stub driver. On successful completion, the stub driver sends back the URBs to the virtual host controller. The communication between the virtual host controller on the client and the stub driver on the server is tunnelled through a TCP/IP connection. Conceptually, USBIP tunnels the communication between a virtual USB bus on the client and a physical device connected to a USB port on the server. As a consequence of this, the USB device is required to remain connected for the protocol to work. If an extended device is disconnected and again connected, the user has to re-establish the tunnel as the previous connection gets lost. In addition, the protocol does not take security into consideration.

3.3 Modbus TCP/IP

A small number of proposals exist for extending industrial field buses over networks. The Modbus [8] protocol is a messaging standard developed for industrial applications by Modicon to establish a client/server communication between devices connected to a serial bus. It has a master/slave architecture. At a time, there is only one master, which initiates the request. The request contains the address of the intended device. Slave devices listen for requests. The target device accepts the request and processes it. The protocol is very simple and has become very popular in industrial settings.

Modbus TCP/IP [9] extends the Modbus protocol over networks. This enables connectivity among devices that are not on the same serial bus but are connected over TCP/IP networks. The Modbus protocol data units are encapsulated inside TCP/IP packets. A special header is added, which is used to differentiate Modbus TCP/IP packets from Modbus packets. Additionally, length information must be kept, since networks may split the packet. In a typical Modbus TCP/IP deployment, a pair of *gateway nodes*, one at the server and one at the client, perform the necessary translation between Modbus and Modbus TCP/IP protocols. The protocol is device, application and operating system independent. Limitations of protocols like Modbus TCP/IP are twofold primarily. First, the protocol allows for static tunneling between a predefined server and a predefined device. No device discovery or other form of dynamic configuration is supported. Second, Modbus TCP/IP lacks mechanisms to secure the communication between servers and clients.

3.4 USB Sensor Network for Industrial Applications

The simplicity, ”plug & play” feature, and power supply to devices by bus are some of the special properties of USB. These are also reasons why many sensors and actuators have been developed with USB interfaces. Power source from the bus, for example, reduces the cabling cost to power the devices. A micro controller with USB interface and Network interface is used to create a USB sensor network in industries [10]. The software in microcontroller has two main modules: network and USB. The network module is used to connect with other users in the LAN. The USB module is used to communicate with the device. The microcontroller does not process the data. It simply submits data to or receives data from the USB device. The network module has a Modbus TCP protocol implemented, which bridges data between USB and TCP.

3.5 iPCI

Since late 1990, Peripheral Component Interconnect (PCI) [11] has become popular as a way to provide an abstraction of processor bus to devices connected to it. It provides high bandwidth and low overhead quality of service (QoS). PCI was the first bus to support plug-and-play and automatic configuration. More recently, an architecture and a protocol satisfying QoS requirement of PCI over networks [12] have been proposed in the literature. In this protocol, PCI transport packets are encapsulated into network packets for transportation between bus and device over an IP network. A light-weight and low-overhead protocol is described for making PCI devices available over a network. The protocol is operating-system independent, device independent, and does not rely on particular application level software. There are three types of connectivity mode described:

1. *IP-based Network Packet Mapping*: In this mapping, the PCI transport packets are encapsulated within TCP, IP and Ethernet headers and footers. It has least data throughput but maximum flexibility in the quantity and distribution of I/O resources.

2. *Switched LAN Packet Mapping*: In this mapping, the PCI transport packets are encapsulated within Ethernet header and footer. It provides more throughput than IP-based but less than Direct Connection. It provides access to PCI bus of machines on local network.

3. *Direct Physical Connect Packet Mapping*: In this mapping, the PCI transport packets are encapsulated within physical layer headers. It has maximum throughput but least flexibility. Only one remote PCI bus per Ethernet port can be accessed.

As there is no implementation of this proposed protocol for PCI over networks, there is no empirical evaluation of the impact on either performance or QoS.

3.6 Wireless USB

Wireless USB [13] provides support for connectivity of USB devices to a computer in a wireless setting. It uses Ultra-wideband (UWB) radio platform to support high bandwidth wireless connectivity between a wireless USB device and a host. Its range is very limited around 10m. With such low range, the computer has to be very close to the device. The scope of extension is limited.

3.7 Xively

Xively [14] is an infrastructure to build applications using a variety of connected devices. For example, an application can be created that utilizes data from two different sensors. These sensors need not be connected to the machine on which the application is running. A datastream is created from each sensor device to the Xively server. The datastream is utilized by other applications connected to the Xively server. It provides a publish/subscribe infrastructure to publish and utilize data from a device over internet. A bidirectional channel is established between the host connected to the device and the Xively server. A unique API key is used to identify a device. It is actually a datastream that is connected to the Xively server. Data-based triggers can be created on the Xively server to send messages to other web services. Xively only allows data to be shared. Full control of device is not possible. There is no direct communication between the device and the application.

There are two different connections, one from the device to the Xively server, another from the Xively server to the application. Xively can not be used for device controls.

4. USB DEVICE MODEL

In this chapter, we will briefly discuss the USB device model. We will describe the process of device discovery and management and the process of data flow between the USB device driver and the USB device in Linux.

The initial aim of the USB interface was to provide a solution to a mixture of connection methods to PC like serial port, parallel port, keyboard, mice connection and others. The basic structure of USB architecture is based on a tiered-star topology as shown in Figure 4.1. In this topology, there is a *host controller*, connected to a *root hub*, which in turn may have several connection points, called ports. Each port can accept a URB device or a further USB hub. A root hub is a USB hub that is embedded with the USB host controller. By connecting hubs to ports, the number of ports that are supported by the host controller can be easily scaled.

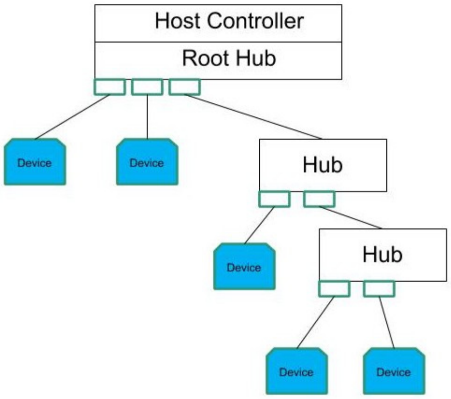


Figure 4.1: USB Topology

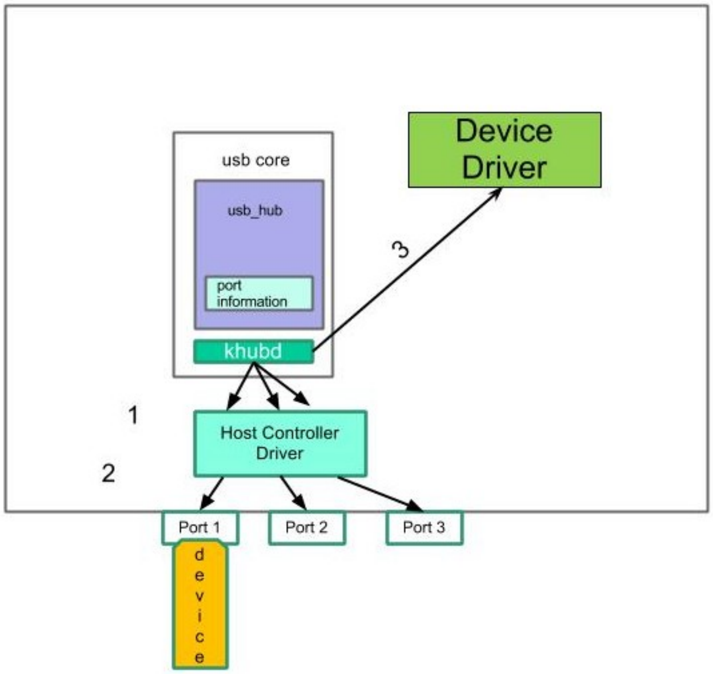


Figure 4.2: Control Plane of USB. Device Insertion: 1. The host controller detects the device insertion by polling. 2. On detecting the device, the device description is retrieved and the device is reset and initialized. 3. The device driver is loaded.

The USB device management is controlled by the so-called *USB core driver*, which in turn controls three aspects of the USB operation: First it handles the detection of events on physical ports. Second, it handles the loading and unloading of device drivers when needed. Third, it dispatches the communication between device drivers and the device. Figure 4.2 describes the control plane of the USB device model. It displays how the device management is done in USB. For each hub, the USB core driver keeps a *usb_hub* data structure. A process, named *khubd* monitors the changes on any port by polling each port of all hubs to check for events at any of the ports. Such events can be the insertion and removal of a device. Depending on the event, a corresponding action is taken. In case of insertion, a device driver loading module is called. In case of removal, a device driver removal module is called.

Whenever a device is connected to a port, for example, the *khubd* process identifies the hub and the port at which the event took place. It then retrieves the device description from the device itself by sending it a specially crafted control USB request block (URB). The device replies by filling the URB with the description of the device. After receiving the descriptor, the USB core driver resets the device and assigns it a unique address. This is called *device enumeration*. Then, the USB core driver uses the device information to find an appropriate device driver for the device. Whenever the USB core communicates with a device, be it to send or receive data, or to control the device, it does so by sending and receiving URBs. Whenever a URB is sent, it reaches all of the devices connected to a hub. Since each device knows its address, assigned during device enumeration, it matches the destination address in the URB and accepts the URB if the address matches. Similarly, when a device is disconnected, the khubd process identifies the hub and port and unload the device driver that was bound to the device on that port. This process informs the device driver about the removal of device.

4.2 Data Communication

Figure 4.3 describes the data communication module of the USB device model. All communications on a USB bus are initiated by the host. Since a device cannot initiate a communication, the device transfers data only when requested by the host. Whenever a device driver wants to read data from a device or to write data to a device, it creates a URB of type IN or OUT respectively and submits it to the

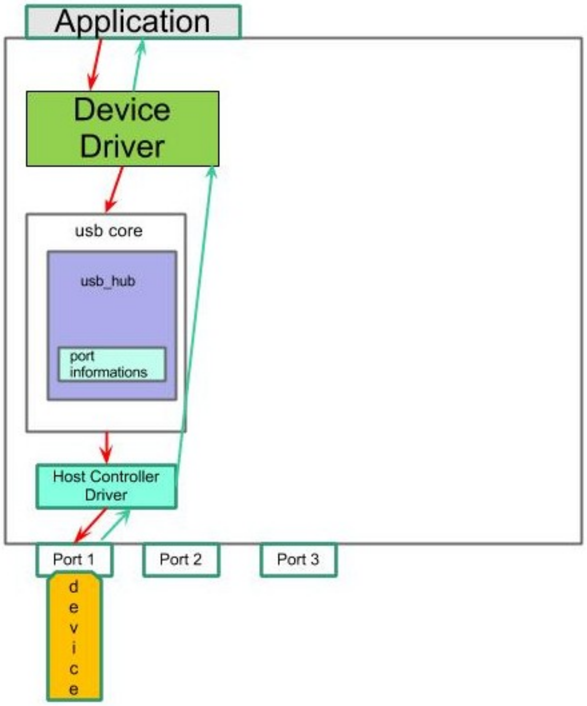


Figure 4.3: Data Plane of USB. The data request is initiated from the application. The device processes the request and replies back with a response

USB core driver. The USB core driver in turn finds the *host controller driver* (HCD) attached to the corresponding hub and submits the URB to it. The HCD, depending on the URB type, submits the URB to the device. The device driver registers a callback method, which is called back by the HCD when the URB is successfully processed by the device. For an IN URB, the device puts the requested data into the URB, which is later read by the device driver. For an OUT URB, the device reads data from the URB transfer buffer and processes it accordingly.

5. USB PORT REMOTING

In this chapter, we will discuss the design and the protocol of a practical example of remote USB ports. We will discuss the design of how to encapsulate the USB device model over IP. This will entail the design of the port management (how to mark ports available to remote hosts), the device discovery (what happens when devices are connected to or disconnected from ports), and the data exchange between the remote client and the device. Finally, we will discuss the security model.

As described in Section 1, our objective is to provide transparent remoting of USB ports. In order to accomplish this, the device discovery and device driver management on the client must not appear different than in the local-device case. In our approach to USB port remoting, the objective is to follow the semantics of traditional, i.e. non-remoted, USB. That is, whenever a device is inserted, the device driver loading module is called, and whenever a device is removed, the device driver using that device is informed and unloaded. The data module also follows the same semantics: the USB device driver submits the URB to the USB core driver. The USB core driver submits it to the host controller driver. Finally, the host controller driver submits it to the device.

In our implementation, the remote port is accessed at the client through a *proxy host controller*, which represents a *proxy hub*, which in turn hosts *proxies* of *remote ports*. Instead of sending URBs to and from the local device, the device driver communicates with the proxy host controller, which in turn exchanges URBs with the proxy port. The proxy port tunnels the communication to a *stub driver* at the server. To the devices connected to the server, the stub driver acts as proxy for the device driver (if any) at the client.

5.1 Port Management

On the server machine, physical USB ports can be made available to remote machines as remote USB ports. A local USB port can be in any of the following states:

1. *Enabled*: This means that when any device is inserted in the port, then a device driver appropriate for that device is loaded. This is the default state.

2. *Disabled*: The USB port is not monitored by the hub controller. When a device is inserted in the port, no device driver is loaded.

3. *Remoted*: A remoted port can be imported by a client. If any device is inserted in the port, then a *stub driver* is loaded, which is capable of communicating with a remote device driver over networks.

Port management handles the status of ports (enabled, disabled, remoted) on the server side. On the client side, port management enables ports to be imported. Imported ports are represented on the client by corresponding *proxy ports*, which tunnel communication from the client device driver to the remote port. Figure 5.1 illustrates the port management.

5.1.1 Server

The port status is maintained in the *port status table*, which is stored with each USB hub data structure. The status of ports are maintained in this table. By default, all ports in an USB hub are enabled and work as local ports. When the user disables a port, the port status table entry corresponding to the port is marked as *disabled*. If there is already a device in that port, then the corresponding device driver is unloaded. No more device drivers are loaded for any device inserted in the

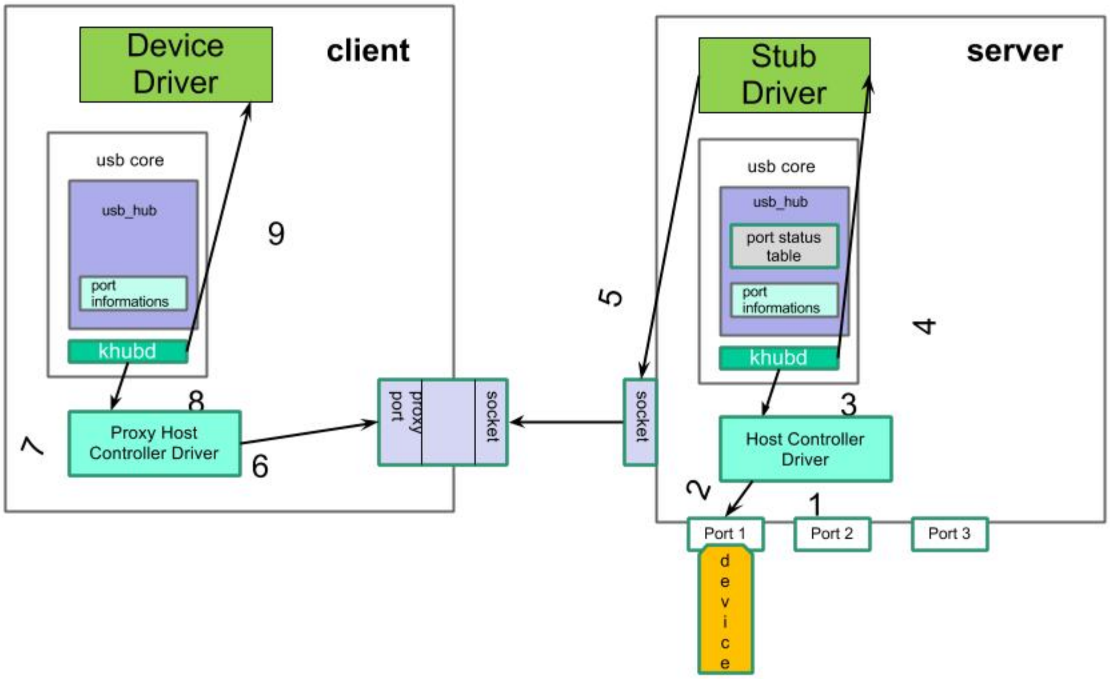


Figure 5.1: Control Plane of Remote USB. Device Insertion: 1. The server detects the device insertion by polling. 2. The device description is retrieved. 3. The device is reset and initialized. 4. The stub driver is loaded. 5. The stub driver informs the proxy port about device insertion. 6. The proxy host controller detects the device insertion by polling. 7. Device description is retrieved. 8. The device is initialized. 9. The device driver for the device is loaded.

port until it is enabled again. When a port is *remoted*, the port status table entry corresponding to the port is marked as *remoted*. Whenever a device is inserted into the port, a *stub_driver* is loaded and bound to the device. If a port is remoted that has already a device inserted, the port is unbound from the current device driver and the *stub_driver* is bound to it. In order for a port to be imported on a client, it must be in the remoted state on the server. We notice that there is no need of any specific device driver on the server machine for a device that is attached to a remoted port. Rather, the *stub_driver* is bound to all devices inserted in remote ports.

5.1.1.1 Stub Driver

A stub driver is a special USB device driver on the server, which communicates with a device on an exported local USB port on behalf of the device driver on the client. It receives URBs submitted by the device driver on the client and submits them to the device on the exported USB port on the server. When the URB is processed by the device, the stub_driver sends the processed URB to the client. When a device is inserted in a remoted USB port, but not exported, the stub_driver is only loaded and does not do anything unless some client imports that port. When the remoted port is also exported, the stub_driver is the entity on the server responsible for informing the client that a device is present on the remote USB port.

5.1.2 Client

On the client machine, remote USB ports are imported. There is a special host controller the *proxy host controller*, which provides communication between the device on the remote USB port and the device driver on the client. The proxy host controller is attached to a proxy hub. The proxy hub in turn maintains information of proxy USB ports. A proxy USB port per remote USB port is maintained to communicate over networks to the remote USB port on the server. From the proxy host controller's view, a proxy USB port looks identical to a USB port. The USB core polls the proxy USB port through the proxy host controller for any event. Whenever a device is inserted in a remote USB port, the USB core on the client finds it out by polling the proxy port, attached to the remote USB port. Similarly, whenever a device is removed from a remote USB port, the USB core on the client finds it out by polling the proxy port, attached to the remote USB port. The USB core on finding the events, triggers the device driver loading or unloading module accordingly.

5.1.3 usbPortManager

The *usbPortManager* is a user level API to manage local ports in the server and proxy ports in the client. This interface takes commands and controls the port status table for managing local ports in the server. In case of proxy ports, the *proxy host controller* driver modifies the proxy port. It is provided through user level commands. There are two sets of commands: one for local ports and the other for proxy ports.

5.1.3.1 Commands for Local Ports

In response to the commands issued for local ports, the port status table is modified. Except the *list* command, all other commands require a port address. The commands are:

- *enable*: This command marks the port *enabled.*
- *disable*: This marks the port *disabled.*
- *remote*: This marks the port *remoted.*
- *unremote*: This marks the port not *remoted.*
- *list*: This lists the local USB ports and drivers bound to devices in those ports.

5.1.3.2 Commands for Proxy Ports

These commands are for managing the proxy ports. All of the client commands take server address and client authentication parameters. The commands are:

- *list*: It lists the ports available in the server, which can be imported.
- *attach*: It takes a server port address as an additional parameter. If the port is available, a proxy port is created on the client.

- *detach*: It takes a proxy port address as an additional parameter. From the proxy port address, the remote port address and the server information is retrieved. The server marks the port available and the proxy port is destroyed on the client.

5.1.3.3 Server Daemon

A user level daemon process running on server machine handles requests from clients. It processes only two types of requests:

- *import*: If the requested port is available, then the client information is stored in the port status table.

- *release*: If the client making the request has imported the requested port, then the port is marked available for further import request and the client information is deleted from the port status table.

5.2 Data Module

This module handles the data communication between a device and a device driver. The data flow path for the local USB device and its driver is from the device driver to the USB device via the USB core driver and the host controller driver. In case of remoted ports on the server, the device driver is at the client machine. The device driver creates URBs and submits them to the host controller of the proxy hub which transfers them to the corresponding proxy USB port which is connected to the server side stub driver over networks. The stub driver at the server side is like a device driver which submits the URBs to the local host controller driver, which in turn submits them to the real device. The return path is similar, the URB processed by device is sent from the host controller on server side to the proxy host controller on the client side via the stub driver and the proxy port connected over socket. We use

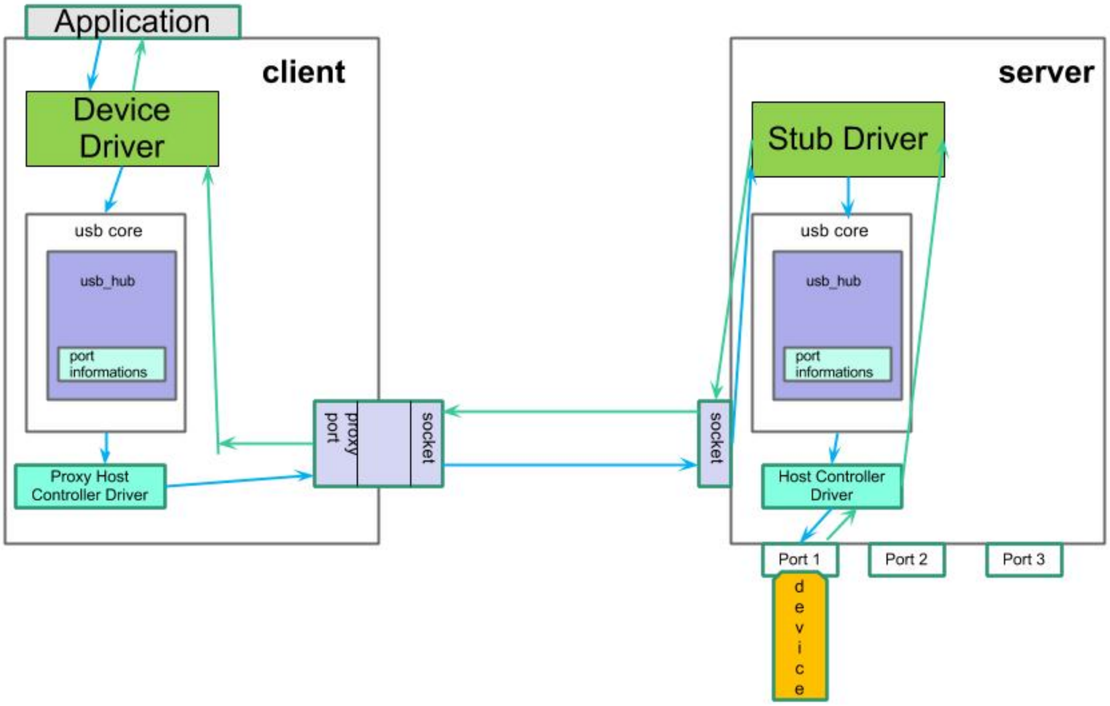


Figure 5.2: Data Plane of Remote USB. The data request is initiated from the application. The device processes the request and replies back with a response

a regular ping mechanism to keep the connection alive and also to find any network failure. Figure 5.2 displays the data module.

5.2.1 Encapsulation of URBs

In order to exchange URB between server and client, the URBs have to be transferred over networks. The URBs are encapsulated in protocol data units. Both server and client transfer the URB after encapsulating it inside a protocol data packet. The client encapsulates the URB submitted by the device driver and sends it to the server. The server decapsulates the URB from the received protocol packet and submits it to the device in local port. The server on receiving the processed URB, encapsulates it again in a protocol packet and sends it to the client. The client decapsulates the

URB from the protocol packet and sends it to the device driver. While encapsulating, serialization of the URB is done and while decapsulating, deserialization of the URB is done. The encapsulation of URBs are similar for all traffics except for the isochronous traffic. In isochronous traffic, the URB has several children packets. So they also are serialized while encapsulating and deserialized while decapsulating.

Figure 5.3 displays the format of a protocol packets. There are two types of protocol packets, data and control. The data protocol packets are used to transfer URBs between server and client. The control protocol packets are used in transferring control messages. The data packets have all the three parts: basic header, command specific header and the URB. In case of control packets, a basic header is present. The command-specific header is present in some control messages. The URB is not present in case of control messages. The basic header contains basic protocol information: sequence number, type of request, remote USB port address, the direction of transfer and the endpoint number. The command-specific header contains information depending on the command. A URB submit command contains the length of the URB in this header. A *import* command stores the port address in this header. The URB in protocol packet is in serialized form.

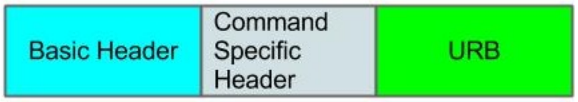


Figure 5.3: Packet Format

5.3 Security

We expect that USB can be remoted over long distances, that is, beyond premises that can be secured. Therefore, the USB remoting protocols must ensure the integrity and confidentiality of the URBs exchanged between client and server. We are providing the user authentication and confidentiality of protocol headers. The payload confidentiality and integrity is not provided. The integrity of payload can be provided by storing the hash of the payload in the protocol header.

For authentication, the server stores all the legitimate users along with their passwords. The import request has port address encrypted with the client's password as key and userid in plaintext. The server retrieves the client's password from the userid in the import request and use it to decrypt the port address.

The confidentiality of the protocol header is maintained by encryption. The server and client both encrypts the protocol header of the protocol packets before sending them over networks. The encryption is done with the password of the client as key. For decryption also the same key is used.

6. EVALUATION

In this chapter we will discuss the implementation and evaluation of our solution. First we will describe the implementation of port status table, the stub driver, the proxy port and the security. Finally, we will evaluate our solution.

6.1 Implementation

We implemented our solution in Linux 3.5 kernel. The USB core driver is modified to store port status table. An user interface *usbPortManager* is used to modify them. We modified the USBIP drivers to support extension of a USB port over networks. There were two types of codes. The driver changes were made in kernel. An user level interface for managing the physical USB ports and proxy USB ports is implemented. An user level daemon is implemented for server machines to process client requests.

6.1.1 Kernel Level Implementations

At kernel level, the port status table is implemented in the USB core driver. The proxy port and the stub driver are implemented in the USBIP drivers.

6.1.1.1 Port Status Table

The port status table stores the status of all the ports of a hub. The USB core driver stores *usb_hub* data structure for each hub. This data structure stores the various configuration parameters of a USB hub and information about all the ports on the hub. In this data structure, we have implemented the port status table. It is implemented with two bitmaps and an array. The bitmap has one bit information for each port and the array has a field for each port. The *disabled_ports* bitmap tells whether the port is disabled or not. By default all bits are zero i.e. enabled. The *remoted_ports* bitmap tells along with the *disabled_ports* bitmap that whether the port is remoted or not. The array *client_info* stores the socket descriptor to communicate with the client and the key of the client for encryption/decryption.

We used *sysfs* mechanism to pass the information from user level to kernel level. A new device attribute *manage_port* is created for the hub device. To manage the port status table, the data is passed to the attribute *manage_port* in the form ”$< command >< portaddress >< arg1 >< arg2 >$”. For enabling, disabling, remoting and unremoting the ports, arg1 and arg2 are blank. When a port is exported to a client, the arg1 is socket file descriptor of the socket connected to the client and arg2 is the key of the client.

6.1.1.2 Stub Driver

The stub driver is a special device driver that can communicate over networks. Whenever a port is imported by a client, the server passes the client information to the stub driver, bound to the device inserted in the port. The client information is its key and socket to communicate with the client. The stub driver starts two threads, one for transmitting messages and the other for receiving messages. The first message sent to the client is the *device inserted* message. In response to this, the client starts device initialization process. Whenever the device is removed, the threads are stopped and a last message *device removed* is sent to the client. When no client has imported the device, no threads are running. The device driver does nothing in that case. This driver has an attribute *usbip_sockfd*. It is used to pass the client socket and key information to the driver from the user level daemon *usbPortManagerD*.

6.1.1.3 Proxy Host Controller

The proxy host controller is a host controller driver. It receives the URBs from the USB core driver and passes them to the proxy USB ports to send them to the server. It also creates and destroys proxy USB ports. It is bound to a proxy hub device.

Every host controller driver has to register a *hc_driver* type of variable with the USB core. The *hc_driver* has two function pointers, *urb_enqueue* and *urb_dequeue*. These function pointers are used by the USB core driver to submit URBs to device or unlink URBs from device respectively. The proxy host controller on receiving the URBs through these enqueue and dequeue function pointers, passes them to the corresponding proxy ports. For creation and destruction of proxy ports, two proxy hub device attributes *attach* and *detach* are used. The server connection information and client key are used to create proxy ports. The *detach* attribute is used to destroy the proxy port.

6.1.1.4 Proxy Port

The proxy ports are analogous to physical USB ports. The proxy port sends the data to the device on the server and receives data from the device on the server. It informs the client USB core when a device is inserted or removed from the remote USB port. It is implemented with two threads, one for sending data to the server and the other for receiving data from the server. It handles two types of protocol packets: data packets and control packets. The data packets encapsulate the URBs, being exchanged between the device driver on the client and the device on the server. The control packets are for informing when a device is inserted or removed. When a device is inserted, the USB core driver on the next poll to the port detects it and starts the device initialization process using data protocol packets. When a device is removed, the proxy port on polling informs the USB core that the device is removed and in turn the device driver is unbound.

The proxy hub attached to the proxy host controller driver manages the proxy ports. It has two device attributes *attach* and *detach*. Whenever the client wants to import a remote USB port of the server, the *usbPortManager* is used to attach to the remote USB port of the server. If the port is available, the socket information and client key is passed to the *attach* attribute. The proxy host controller driver creates a proxy USB port with this socket and key information. The transmitting and the receiving threads are started using the socket information. Whenever the client wants to cancel the import of any remote USB port, the *usbPortManager* is used to detach from the remote USB port of the server. The proxy host controller is passed this information through *detach* attribute and the server is informed. The proxy host controller in turn stops the transmitting and the receiving thread and destroys the socket. Thus the proxy port is destroyed.

6.1.1.5 Security

We are providing the user authentication and the protocol header confidentiality feature. For authentication, the client user id and key is stored in the server. The import request has two fields: the user id and the port address. The user id is in plaintext and the port address is encrypted with the key of the client. The server on receiving an import request, retrieves the user id from the request. The user id is used to retrieve its password. The password is used as key to decrypt the port address. The password is stored in the port status table along with the socket information of the client.

For confidentiality of protocol headers, the server and the client both encrypts the protocol header before sending it over networks. The key used in the encryption in case of client is stored in the proxy USB port. In server, the key is stored in the port status table. The *stub_driver* retrieves the key from the port status table and use it to encrypt and decrypt the protocol headers of each packet.

6.1.2 User Level Implementations

At user level, there are two modules. The *usbPortManagerD* daemon to process the clients' requests and the *usbPortManager* utility to manage the physical USB ports on the server and proxy USB ports on the client.

6.1.2.1 usbPortManagerD Daemon

A daemon, *usbPortManagerD* runs on the server to receive *import* and *release* requests from the clients. It listens on a port for incoming connections. It receives the request, authenticates the client and if the request is valid, corresponding actions is taken. In case of *import* request, the port address is retrieved from the request. If the port is available for import, the client is passed a success message and the client information (socket and key) is passed to the USB core through the *match_port* attribute of hub using *sysfs* libraries. If a device is already connected in that port, then the client information is also passed to the device driver using its attribute *usbip_sockfd*.

6.1.2.2 usbPortManager

usbPortManager is used to manage physical USB ports in the server and proxy USB ports in the client. It is implemented using socket and sysfs libraries to communicate with server and the kernel level attributes.

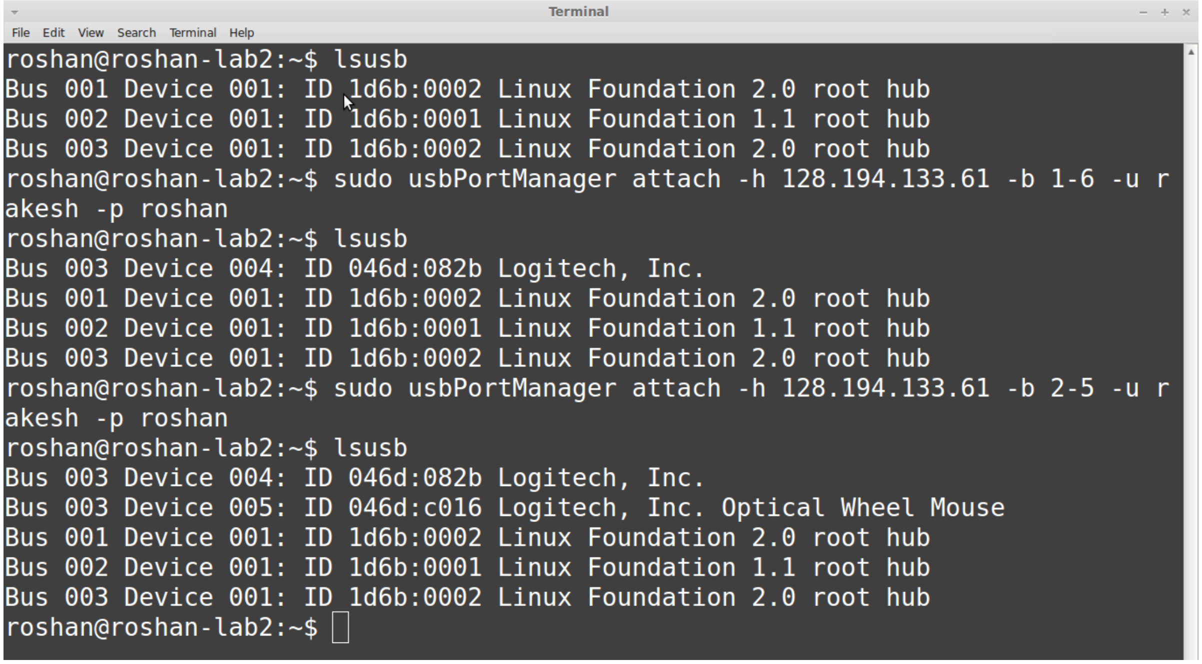
For managing physical USB ports, the hub's *manage_port* attribute is used to manage the ports. It is used to mark the port enabled, disabled or remoted. To mark the port with any status, the message corresponding to the status is passed to the *manage_port* attribute. In turn, the port status table entry for the requested port is modified.

For managing proxy USB ports, first of all, a connection is established with the server. Then the import request is sent. If the requested port is available, then the message is sent to the proxy host controller using the proxy hub's attribute *attach*. In response to this, the proxy host controller creates a proxy USB port. Similarly, when releasing the imported port, the proxy hub's *detach* attribute is used to destroy the proxy USB port.

6.2 Results

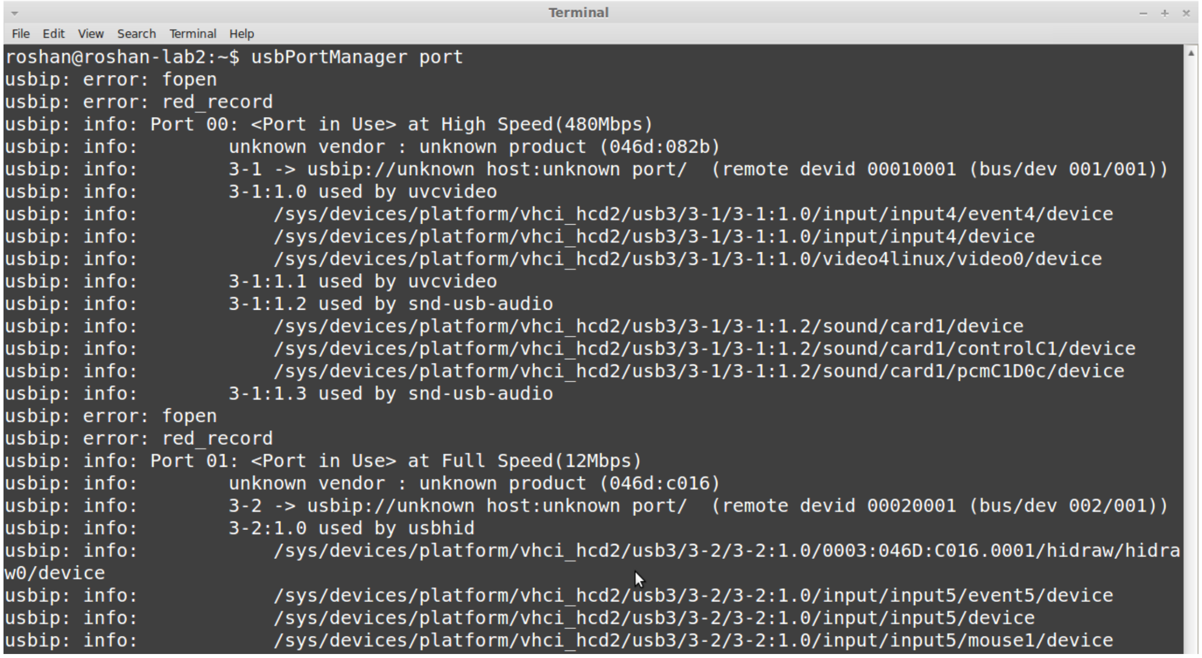
We set up a number of experiments to assess to which level, the solution criteria laid out in the Section 2 have been satisfied by our implementation. We performed experiments on a USB mouse, a USB webcam, a USB mic, a USB flash drive and a USB hard drive. We are describing our evaluation for each of the criteria.

6.2.1 USB Protocol Independent



```
roshan@roshan-lab2:~$ lsusb
Bus 001 Device 001: ID 1d6b:0002 Linux Foundation 2.0 root hub
Bus 002 Device 001: ID 1d6b:0001 Linux Foundation 1.1 root hub
Bus 003 Device 001: ID 1d6b:0002 Linux Foundation 2.0 root hub
roshan@roshan-lab2:~$ sudo usbPortManager attach -h 128.194.133.61 -b 1-6 -u r
akesh -p roshan
roshan@roshan-lab2:~$ lsusb
Bus 003 Device 004: ID 046d:082b Logitech, Inc.
Bus 001 Device 001: ID 1d6b:0002 Linux Foundation 2.0 root hub
Bus 002 Device 001: ID 1d6b:0001 Linux Foundation 1.1 root hub
Bus 003 Device 001: ID 1d6b:0002 Linux Foundation 2.0 root hub
roshan@roshan-lab2:~$ sudo usbPortManager attach -h 128.194.133.61 -b 2-5 -u r
akesh -p roshan
roshan@roshan-lab2:~$ lsusb
Bus 003 Device 004: ID 046d:082b Logitech, Inc.
Bus 003 Device 005: ID 046d:c016 Logitech, Inc. Optical Wheel Mouse
Bus 001 Device 001: ID 1d6b:0002 Linux Foundation 2.0 root hub
Bus 002 Device 001: ID 1d6b:0001 Linux Foundation 1.1 root hub
Bus 003 Device 001: ID 1d6b:0002 Linux Foundation 2.0 root hub
roshan@roshan-lab2:~$
```

Figure 6.1: Importing Remote USB Ports



```
roshan@roshan-lab2:~$ usbPortManager port
usbip: error: fopen
usbip: error: red_record
usbip: info: Port 00: <Port in Use> at High Speed(480Mbps)
usbip: info:        unknown vendor : unknown product (046d:082b)
usbip: info:        3-1 -> usbip://unknown host:unknown port/  (remote devid 00010001 (bus/dev 001/001))
usbip: info:        3-1:1.0 used by uvcvideo
usbip: info:            /sys/devices/platform/vhci_hcd2/usb3/3-1/3-1:1.0/input/input4/event4/device
usbip: info:            /sys/devices/platform/vhci_hcd2/usb3/3-1/3-1:1.0/input/input4/device
usbip: info:            /sys/devices/platform/vhci_hcd2/usb3/3-1/3-1:1.0/video4linux/video0/device
usbip: info:        3-1:1.1 used by uvcvideo
usbip: info:        3-1:1.2 used by snd-usb-audio
usbip: info:            /sys/devices/platform/vhci_hcd2/usb3/3-1/3-1:1.2/sound/card1/device
usbip: info:            /sys/devices/platform/vhci_hcd2/usb3/3-1/3-1:1.2/sound/card1/controlC1/device
usbip: info:            /sys/devices/platform/vhci_hcd2/usb3/3-1/3-1:1.2/sound/card1/pcmC1D0c/device
usbip: info:        3-1:1.3 used by snd-usb-audio
usbip: error: fopen
usbip: error: red_record
usbip: info: Port 01: <Port in Use> at Full Speed(12Mbps)
usbip: info:        unknown vendor : unknown product (046d:c016)
usbip: info:        3-2 -> usbip://unknown host:unknown port/  (remote devid 00020001 (bus/dev 002/001))
usbip: info:        3-2:1.0 used by usbhid
usbip: info:            /sys/devices/platform/vhci_hcd2/usb3/3-2/3-2:1.0/0003:046D:C016.0001/hidraw/hidra
w0/device
usbip: info:            /sys/devices/platform/vhci_hcd2/usb3/3-2/3-2:1.0/input/input5/event5/device
usbip: info:            /sys/devices/platform/vhci_hcd2/usb3/3-2/3-2:1.0/input/input5/device
usbip: info:            /sys/devices/platform/vhci_hcd2/usb3/3-2/3-2:1.0/input/input5/mouse1/device
```

Figure 6.2: Details of Devices Connected to Remote USB Ports

In order to illustrate the USB protocol independence of our implementation, we show how we can successfully import devices that use different USB protocols (1.1, 2.0 in our case). Specifically, we connected a USB webcam that was a USB 2.0 device, while also a USB 1.1 mouse. The webcam utilized isochronous traffic to exchange data. The mouse utilized interrupt traffic to exchange data. The control traffic is used in all USB devices in device initialization. Two remote USB ports were imported by the client machine. On one port, the mouse was connected. On second port, the webcam was connected. We used a Kamoso application to test the webcam. The Matte desktop manager was used to test the mouse. The Kamoso application displayed video smoothly. The mouse cursor was also moving freely. We took the screenshot of the screen which displays the details of the devices connected to the proxy ports and their details. The Figure 6.1 is the screenshot of importing remote USB ports at the client machine. It shows two devices. One is a Logitech Mouse and the other is a Logitech webcam. In the Figure 6.2, the details of the devices connected to the imported ports are shown. The mouse was a Full Speed (USB 1.1) device and the webcam was a High Speed (USB 2.0) device.

6.2.2 Device Agnostic

In order to illustrate the device agnosticism, we show how we can successfully connect different USB devices to the remote USB ports. We connected these USB devices to local USB ports too. The devices were working and applications used the devices successfully irrespective of the devices connected on local USB port or remote USB port. We connected a Logitech USB webcam, a Logitech USB mouse as shown in the Figure 6.1 and used applications Kamoso for webcam and Mate window manager for mouse. We saw that device was not aware, whether it was being accessed remotely or locally. They worked same in both cases.

6.2.3 Device Driver Independent

In order to illustrate the device driver independence, we show how we can successfully bind same device driver for devices connected to the local USB port or the remote USB port. In Linux 3.5, the device driver for a USB webcam is *uvcvideo*. The device driver for a USB mic is *snd-usb-audio*. The device driver for a USB mouse is *usbhid*. We first connected all the devices to a local USB port. Then we connected all of them to a remote USB port. In both cases, the corresponding drivers for the connected devices were loaded irrespective of the USB port types. In Figure 6.2, we see the device drivers bound to the mouse and webcam connected to the remote USB ports. We also used applications utilizing those device drivers like Kamoso for camera and Mate for mouse. The same device driver worked same whether the device was in the local USB port or the remote USB port.

6.2.4 Performance Transparency

We are extending a port over network. We have to maintain a performance transparency so that, the applications utilizing the device on those ports are not affected. In this test, we are testing at two levels. System level and Application level. At system level, we measured the end-to-end latency for each URB when submitted to a local USB port device and a remote USB port device. We then compared them. At application level, we used Bonnie++ [15] benchmark tool to test hard drive and file system performance.

We tested on a USB webcam, USB mic and USB Flash Drive. USB webcam and USB mic worked on the isochronous traffic while USB flash drive works on the bulk traffic. In the isochronous traffic, the packets are exchanged at a regular interval. It is given the highest preference by host controller for bandwidth allocation. Bulk traffic is given the lowest preference by host controller while bandwidth allocation. The machines for testing were connected over a 100 Mbps wired connection. We used *usbmon* [16] module to monitor the USB traffic.

6.2.4.1 Isochronous Traffic

The isochronous traffic is used by devices, which need a guaranteed bandwidth. The USB host controller gives the highest priority in bandwidth allocation to the isochronous traffic. We used the USB webcam and the USB mic to test the isochronous traffic performance. The USB mic used packets of size 152 bytes. The USB cam used two different types of URBs, one of size 560 bytes and the other is of size 39345 bytes. The performance is affected by the CPU utilization of the client machine, the CPU utilization of the server machine and the network latency. These factors are always varying. We found the average end-to-end latency in an URB processing. We used the *usbmon* module to find out the end-to-end delay between an URB submission and its completion.

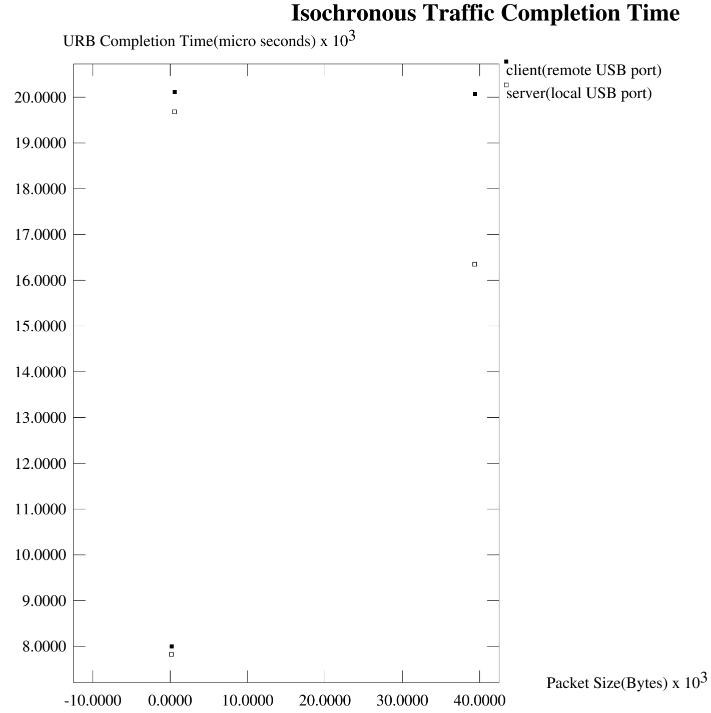


Figure 6.3: URB Completion Time Vs URB Size

In Figure 6.3, the URB completion time of URBs of different sizes on the server machine and the client machine are displayed. The graph in Figure 6.4 displays how the latency in the URB completion time at the client varied with the URB size.

Average completion time of a URB in a traffic from the USB mic device was 7996.26 μs, when the device was connected to the remote USB port. When the device was connected to a local USB port, then the average completion time an URB was 7826.15 μs. The average latency was 170.1 μs. In USB cam, there were two types of URBs. One was of size 560 bytes and another of size 39345 bytes. The average completion time of a URB on a camera connected to the remote USB port for 560-*byte* packet was 20111.42 μs and for 39345-*byte* packet was 20063.78 μs. The average completion time of an URB on a camera connected to a local USB port for 560-*byte* packet was 19681.66 μs and for 39345-*byte* packet was 16350.57 μs. Average delay for the 560-*byte* packet was 429.76 μs and for 39345-*byte* packet was 3713.21 μs.

The URB completion time on the client was measured from the time of submitting a URB to the USB core driver to the time of retrieving it back from the USB core. It includes the delay incurred due to the packaging of a URB in a TCP packet, passing it over network, USB processing time at server, passing response back over network and unpackaging a TCP packet to retrieve the URB. On server side, the device is connected to a local USB port. The completion time on the server was measured from the time of submitting a URB to the USB core driver to the time when the URB gets completed. On client side, there were two types of processing, one was USB processing time(done on server), another was protocol and network processing time(in serializing/deserializing URBs). The server-side latency includes the USB processing time. The rest of the client-side latency comprised of protocol and network processing time. We see in the Figure 6.4, the latency increases as the packet size increases. The network connection takes more time to transfer a larger packet so, the latency increases. The test machines were connected over a 100 Mbps connection. If we assume, we were getting full 100 Mbps bandwidth then, we can find the time required to transfer a packet over the network. The Table 6.1 shows the network latency for URB packets of different sizes. We can observe from the results that as the packet size grows, the network latency becomes major delay factor.

Packet Size	Network Delay	Total Delay
152 bytes	11.6 μs	170.1 μs
560 bytes	42.73 μs	429.76 μs
39345 bytes	3001.79 μs	3713.21 μs

Table 6.1: Isochronous Traffic Latency

6.2.4.2 Bulk Traffic

The bulk traffic is used by devices, which exchange data which are not time-sensitive. For example, a USB flash drive uses bulk traffic to exchange data.

We tested on a USB flash drive. The *dd* command was used to read 100 MB of data from the flash drive. We did only a read test on the flash drive, since, write on a flash memory incur some flash layer delay too. The latency is not constant for the same size URB in case of bulk traffic. In Figure 6.5, we see that there was an average increment of 348.53 μs in the completion time of URBs of same size, submitted contiguously. There was a decrease in completion time for smaller URBs.

The URB size requested by the device driver was different if the device was connected to the local USB port and if it was connected to the remote USB port. The graph in Figure 6.6 shows the URB packet sizes in case the device was on the local USB port. The graph in Figure 6.7 shows the URB packet sizes in case the device was on the remote USB port. In both cases, large and small URBs were being exchanged at regular interval. In case of the local USB port, a URB of size 12280 bytes was followed by two 13-*byte* and 31-*byte* packets. In case of the remote USB port, the 12280-*byte* URB was splitted in several smaller URBs. This split caused a large bust of URBs submitted in a very short interval of time. The graph in Figure 6.8 shows how frequently URBs were submitted in the local USB port device. For the remote USB port device, the submission pattern is shown in Figure 6.9. The Table 6.2 displays the distribution of URBs over different submission rates.

Submission Interval (δ)	Number of URBs	
	Local USB Port	Remote USB Port
$\delta < 10\mu$s	0	81
$10\mu s \leq \delta < 100\mu s$	5	0
$100\mu s \leq \delta < 200\mu s$	46	0
$200\mu s \leq \delta < 300\mu s$	15	1
$300\mu s \leq \delta < 500\mu s$	4	7
$500\mu s \leq \delta < 1000\mu s$	14	5
$1000\mu s \leq \delta < 2000\mu s$	0	1
$2000\mu s \leq \delta$	15	3

Table 6.2: URB Submission Rate in Local and Remote USB Bulk Traffic

Out of 100 URBs, more than 80 URBs were submitted at a gap of less than 10 μs in case of the remote USB port. While, in case of the local USB port, no URBs were submitted within 10-μs duration of the last URB submitted. In case of the remote USB port, URBs need to be transferred over networks. On server, the URBs were processed on a device connected to a local port. We see in the Figure 6.10 that, the frequency at which the URBs were submitted, the URBs got completed at almost the same frequency on server. But they could not be delivered at the same frequency to the client due to network latency. So, the URBs got stored in a queue. With each URB, this network latency got accumulated. In the Figure 6.5, we see that the completion time of a URB on a device connected to a remote USB port increased at an average increment of 349.53 μs. The time required to transfer a 4096-*byte* packets over a 100 Mbps network was 312.5 μs. The increasing latency is due to the network latency.

6.2.4.3 Device Management Performance

The device insertion and removal takes place at the remote USB port. It affects the process time of the device insertion and the device removal on the client machine than that done on a local USB port. We observed the time from when the USB core driver detects the insertion or the removal events to the time when the device driver load or unload process is triggered respectively. The processing time of the device insertion and removal is not a constant time process. We tested for 10 insertion operations and took an average of them. The device insertion process in the remote USB port, on an average took 680.83 ms. The device insertion process in the local USB port on an average took 322.86 ms. The Table 6.3 displays the experimental results. The device removal process did not have much effect on the client side. The

Local USB Port	Remote USB Port
322932	654755
323053	683379
322764	684037
322651	684263
322683	683499
322382	683581
323146	683915
323077	683623
323109	683515
322847	683723

Table 6.3: Device Insertion Time (in μs)

average device removal time on the remote USB port was 16848.63 μs. The average device removal time on the local USB port was 12216.1 μs. The average device removal time on the remoted port on the server was 1736426.67 μs. There is large delay on the server side was due to the timeout mechanism to detect the network failure. This timeout mechanism caused the average delay of around 1.74 sec.

6.2.4.4 Application Performance

Bonnie++ [15] is a well known benchmark tool to test the hard drive and the filesystem performance. We used Bonnie++ version 1.96 for our test. A USB portable hard drive with NTFS filesystem was used for testing. The Figure 6.11 displays the result of Bonnie++. We ran Bonnie++ test first on the hard drive connected to the remote USB port. Then we tested by connecting the hard drive to the local USB port. The bandwidth is the average bandwidth while the latency is the maximum latency. So in the Figure 6.11, we see that there is no direct relation between bandwidth reduction and latency increment. We see that the performance decrement in data operations like read and write was more as compared to control operations like seek.

6.2.5 Failure Transparency

The device drivers should not handle the failures introduced by the remote USB ports. Our solution hides all failures and the machine sees such failure as device removal in the client or port unexported in the server. To test it, a USB webcam was connected to the remote USB port. We created different kinds of failures. Disconnected server from the network. Disconnected client from the network. Power off the machines. The client on detecting failure, informed the device driver that device was removed and destroyed the proxy port. In turn, the device driver was unloaded. The server on detecting failure, assumed that the client is no longer interested in the port and marked it to be available for other clients to import.

6.2.6 Security

It uses encryption for integrity and authentication mechanism for secure access to the remote USB ports. We used a non authenticated user to connect to the remote USB port. The request was rejected by the server. To simulate the situation of someone modifying the packet in between, we changed the encryption key on the client side. The server rejected those packets but did not close the connection. The authentic user continued to work.

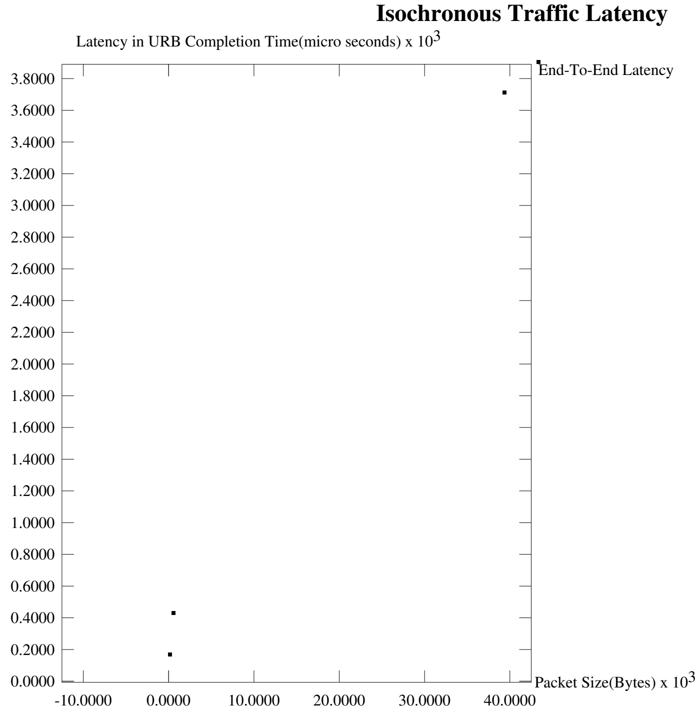


Figure 6.4: Latency in URB Completion Time Vs URB Size

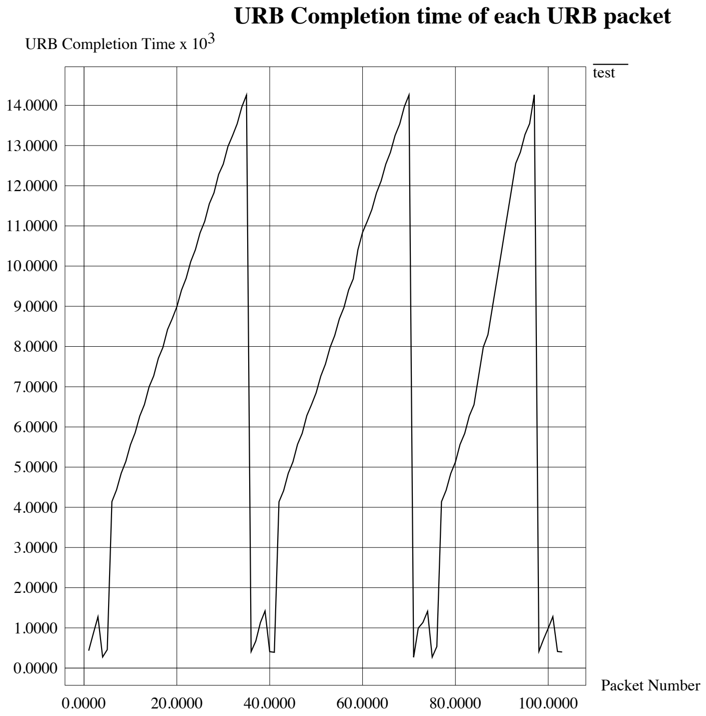


Figure 6.5: Bulk Traffic in Storage Device

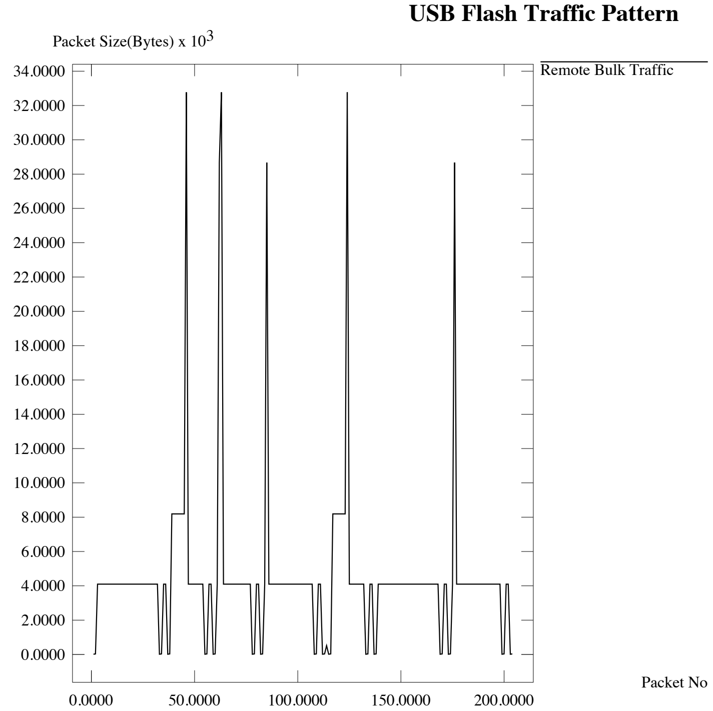


Figure 6.6: Remote Bulk Traffic Packet Size Pattern

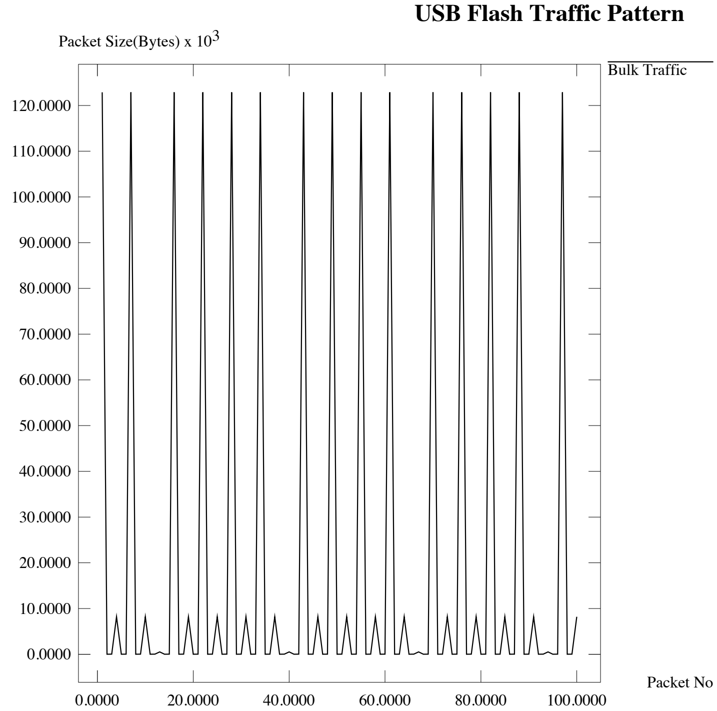


Figure 6.7: Local Bulk Traffic Packet Size Pattern

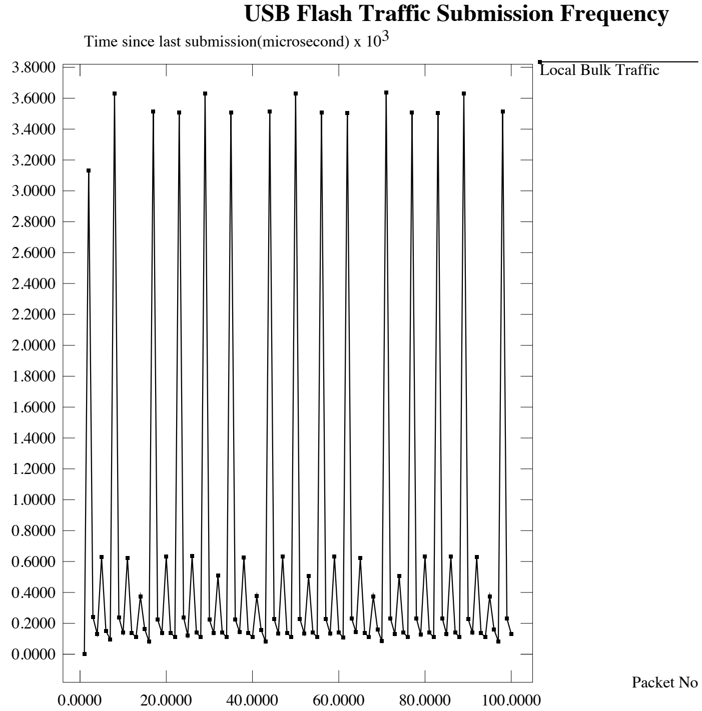


Figure 6.8: Local Bulk Traffic Packet Submission Rate

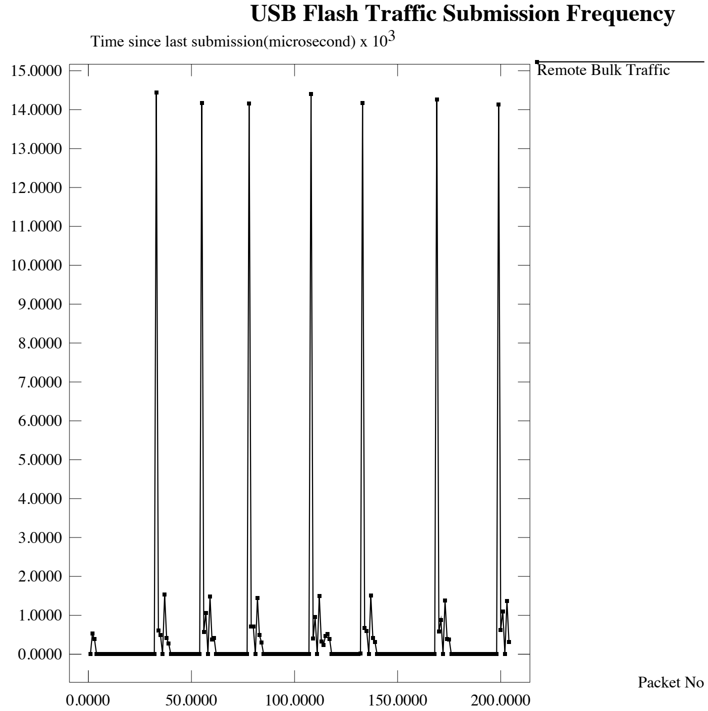


Figure 6.9: Remote Bulk Traffic Packet Submission Rate

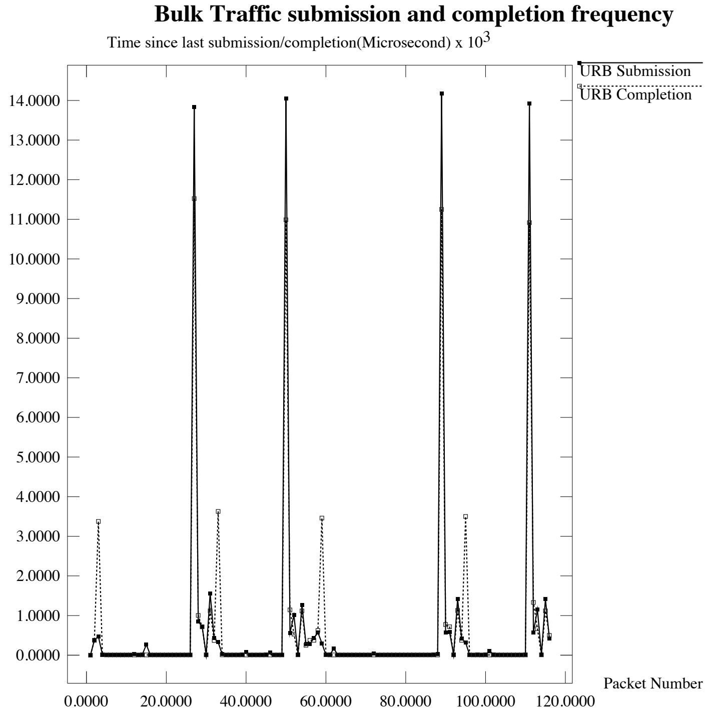
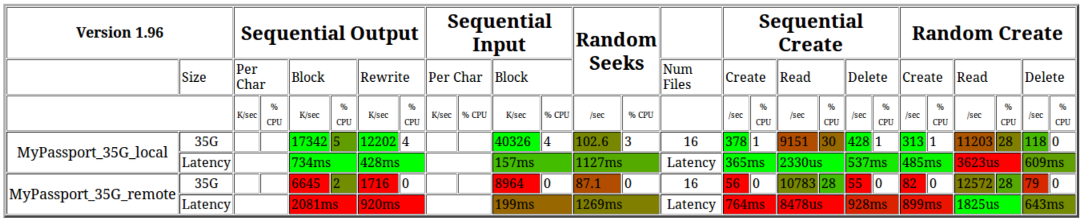


Figure 6.10: Server Bulk Traffic Submission and Completion Rate



Version 1.96		Sequential Output						Sequential Input				Random Seeks			Sequential Create						Random Create					
	Size	Per Char		Block		Rewrite		Per Char		Block				Num Files	Create		Read		Delete		Create		Read		Delete	
		K/sec	% CPU	K/sec	% CPU	K/sec	% CPU	K/sec	% CPU	K/sec	% CPU	/sec	% CPU		/sec	% CPU	/sec	% CPU	/sec	% CPU	/sec	% CPU	/sec	% CPU	/sec	% CPU
MyPassport_35G_local	35G			17342	5	12202	4			40326	4	102.6	3	16	378	1	9151	30	428	1	313	1	11203	28	118	0
	Latency			734ms		428ms				157ms		1127ms		Latency	365ms		2330us		537ms		485ms		3623us		609ms	
MyPassport_35G_remote	35G			6645	2	1716	0			8964	0	87.1	0	16	56	0	10783	28	55	0	82	0	12572	28	79	0
	Latency			2081ms		920ms				199ms		1269ms		Latency	764ms		8478us		928ms		899ms		1825us		643ms	

Figure 6.11: Bonnie++ Benchmark Results

7. SUMMARY

The Remote USB Port provides a remote port which is secure, with low protocol overhead and, device, device driver and operating system independent. For smaller URBs, the protocol overhead is larger as compared to network overhead. But for larger packets, network is the major overhead. We tested on a 100 Mbps connection. Today Gigabit Ethernet is available. With Gigabit Ethernet, network latency will go down. The main protocol overhead is in serialization and deserialization of the packet. The bulk traffic packets take less time in serialization and deserialization. The isochronous URBs have many description fields. The isochronous traffic URBs takes more time in serialization and deserialization. We used symmetric cryptographic technique for encryption/decryption of packet headers. We tested on a 100 Mbps connection. If we test on a Gigabit Ethernet, the latency will go down and bandwidth will increase. On a server machine, all the ports which are remoted use same network interface on which the server daemon is running for communication with client. If there are a lot of remoted ports on the server, the performance will decrease as the same bandwidth will be shared by multiple connections. As of now, the solution is not utilizing multiple interfaces (if available) on a machine. The solution can be extended to utilize multiple interfaces on a machine to provide a better bandwidth in case a server's multiple ports are remoted. Security can be enhanced by using a non-symmetric cryptographic technique.

REFERENCES

[1] N. Falliere, L. O. Murchu, and E. Chien, "W32.Stuxnet Dossier." http://www.symantec.com/content/en/us/enterprise/media/security_response/whitepapers/w32_stuxnet_dossier.pdf, Feb 2011.

[2] IEEE Computer Society, "Part 3: Carrier Sense Multiple Access with Collision Detection (CSMA/CD) Access Method and Physical Layer Specifications," *IEEE Std 802.3TM-2005*, Dec 2005.

[3] USB 3.0 Promoter Group, "SuperSpeed USB from the USB-IF." http://www.usb.org/developers/ssusb, February 2013.

[4] USB 2.0 Promoter Group, "Hi-Speed USB from the USB-IF." http://www.usb.org/developers/usb20, May 2002.

[5] R. O. Weber, "SCSI Architecture Model-3." http://ftp.t10.org/ftp/t10/document.02/02-119r0.pdf, March 2002.

[6] Mosaic Technology, "iSCSI Guide." http://www.mosaictec.com/pdf-docs/whitepapers/iSCSI_Guide.pdf, February 2009.

[7] T. Hirofuchi, E. Kawai, K. Fujikawa, and H. Sunahara, "USB/IP - A Peripheral Bus Extension for Device Sharing over IP Network (Awarded FREENIX Track Best Paper Award!)," in *USENIX Annual Technical Conference, FREENIX Track*, pp. 47–60, USENIX, 2005.

[8] Modbus Organization, "Modbus Application Protocol Specification." http://www.modbus.org/docs/Modbus_Application_Protocol_V1_1b3.pdf, April 2012.

[9] Modbus Organization, "Modbus Messaging on TCP/IP Implementation Guide." http://www.modbus.org/docs/Modbus_Messaging_Implementation_Guide_V1_0b.pdf, October 2006.

[10] A. Depari, A. Flammini, D. Marioli, and A. Taroni, "USB Sensor Network for Industrial Applications," *Instrumentation and Measurement, IEEE Transactions on*, vol. 57, no. 7, pp. 1344–1349, 2008.

[11] Rovin and Sagar, "Introduction to PCI Protocol." http://electrofriends.com/articles/computer-science/protocol/introduction-to-pci-protocol/, October 2009.

[12] D. Daniel and J. Hui, "Virtualization of Local Computer Bus Architectures Over the Internet," in *Global Telecommunications Conference, 2007. GLOBECOM '07. IEEE*, pp. 1884–1889, 2007.

[13] Wireless USB Promoter Group, "Wireless USB from the USB-IF." http://www.usb.org/developers/wusb, September 2010.

[14] LogMeIn Inc, "Xively: Public Cloud for the Internet of Things." https://xively.com/, July 2013.

[15] R. Coker, "Bonnie++." http://www.coker.com.au/bonnie++/, January 2006.

[16] P. Zaitcev, "USB monitoring framework." https://www.kernel.org/doc/Documentation/usb/usbmon.txt, July 2013.