



FACULTY OF INFORMATION TECHNOLOGY AND ELECTRICAL ENGINEERING
DEGREE PROGRAMME IN ELECTRONICS AND COMMUNICATIONS ENGINEERING

MASTER'S THESIS

A remotely accessible USB hub: Software design and testing

Author	Jani Laitinen
Supervisor	Juha Häkkinen
Second Examiner	Kari Määttä

June 2022

Laitinen J. (2022) A remotely accessible USB hub: Software design and testing
University of Oulu, Faculty of Information Technology and Electrical Engineering, Degree Programme in Electronics and Communications Engineering. Master's Thesis, 77 p.

ABSTRACT

Remote use of USB peripherals has been identified as useful for Aava Mobile customers. Therefore, the commercial feasibility of an accessory that allows accessing USB devices remotely was studied at Aava, and a prototype device was built. The software in this accessory was required to transfer data securely, be automatically detectable on a local network, and operate autonomously. It is explored in this thesis how remote USB sharing and the requirements could be implemented using open-source software components.

New USB remote use programs that support the required capabilities were created as part of this thesis. These applications run on Linux-based operating systems and make use of the existing open-source USB/IP tool protocol. The new client program uses the existing Linux USB/IP virtual host controller driver, and the server is implemented in user space.

After the software work was concluded, measurements were performed for evaluation purposes. Optimal encryption ciphers for the prototype hardware were also selected. It was verified by testing that network delay causes major performance degradation. Other significant performance concerns were network adapter speed, the use of encryption, USB port speed, and the user space server implementation. However, while these aspects reduced the performance of the prototype, they were not determined to be critical. The accessory was not intended for high-performance use cases, and therefore the use of cost-effective components can be justified.

Key words: USB/IP, Linux, TLS, remote use

Laitinen J. (2022) Etäkäytettävä USB-keskitin: Ohjelmistokehitys ja testaus Oulun yliopisto, tieto- ja sähkötekniikan tiedekunta, elektroniikan ja tietoliikennetekniikan tutkinto-ohjelma. Diplomityö, 77 s.

TIIVISTELMÄ

USB-laitteiden etäkäyttö on havaittu hyödylliseksi Aava Mobilen tablettilaitteiden käyttäjille. Tästä syystä Aavalla tutkittiin tämän toiminallisuuden sisältävän lisälaitteen kaupallista toteutusmahdollisuutta ja toteutettiin prototyyppilaitte. Tässä laitteessa toimivan ohjelmiston vaadittiin salaavan siirrettävä data, löytyvän automaattisesti sisäverkossa sekä toimivan ilman käyttäjän apua. Tässä diplomityössä tutkitaan kuinka USB-laitteiden käyttö sekä vaaditut ominaisuudet voitaisiin toteuttaa avoimen lähdekoodin ohjelmistokomponenttien avulla.

Diplomityön osana toteutettiin vaaditut ominaisuudet sisältävät ohjelmistotyökalut USB-laitteiden etäkäyttöön. Nämä ohjelmistot toimivat Linux-pohjaisissa käyttöjärjestelmissä ja käyttävät olemassa olevaa avoimen lähdekoodin USB/IP-työkalujen protokollaa. Asiakasohjelma käyttää olemassa olevaa virtuaalista USB/IP-isäntäohjainta ja palvelin on toteutettu käyttäjätilassa.

Ohjelmiston toteutuksen jälkeen mittauksilla arvioitiin suorituskykyä sekä valittiin optimaaliset salausalgoritmit prototyyppilaitteistoa varten. Testeillä vahvistettiin, että verkon viiveellä on suuri vaikutus järjestelmän suorituskykyyn. Muita merkittäviä suorituskykyyn vaikuttavia seikkoja olivat verkkoadapterin nopeus, salauksen käyttäminen, USB-portin nopeus sekä palvelinohjelman toteutus käyttäjätilassa. Nämä hidastivat prototyyppilaitteen toimintaa, mutta niiden vaikutus ei kuitenkaan ollut kriittistä. Toteutettua lisälaitetta ei ollut tarkoitettu käytettäväksi kohteissa, jotka vaativat suurta suorituskykyä ja näin ollen laitteistovalinnoilla voitiin saavuttaa kustannushyötyä.

Avainsanat: USB/IP, Linux, TLS, etäkäyttö

TABLE OF CONTENTS

ABSTRACT

TIIVISTELMÄ

TABLE OF CONTENTS

FOREWORD

LIST OF ABBREVIATIONS AND SYMBOLS

1	INTRODUCTION	8
2	USB SHARING OVER IP	10
2.1	Commercial devices	10
2.2	Commercial software	11
2.3	Open-source USB/IP software	11
2.3.1	Linux tools.....	11
2.3.2	Windows tools	13
2.3.3	Limitations and possible solutions	14
3	INTRODUCTION TO USB	18
3.1	Overview	18
3.2	Data	19
3.2.1	Requests.....	19
3.2.2	Descriptors.....	20
3.2.3	Transfer types	21
3.3	Hardware	22
3.3.1	Host controller	22
3.3.2	Hub	23
3.4	Software in the Linux kernel	23
3.4.1	USB Request Block (URB).....	23
3.4.2	Linux USB stack.....	24
4	USB/IP COMPONENTS AND PROTOCOL	27
4.1	User space applications	28
4.1.2	Protocol and capabilities.....	28
4.2	Linux drivers	32
4.2.1	VHCI driver.....	32
4.2.2	STUB driver	34
4.2.3	Protocol.....	35
5	IMPLEMENTATION.....	41
5.1	VIOBox device.....	41
5.1.1	Single-board computer	41
5.1.2	Mechanical construction.....	42
5.2	Software requirements and design choices	43
5.2.1	Security.....	44
5.2.2	Automatic discovery and unattended use	45
5.3	RemoteHub library and application design	45
5.3.1	Libraries.....	47

5.3.2	Applications.....	52
5.3.3	Licenses	56
5.4	RemoteHub usage.....	57
5.5	USBIP-win software.....	59
6	RESULTS	60
6.1	Optimal environment.....	61
6.2	VIOBox device.....	63
7.	DISCUSSION.....	66
8.	SUMMARY.....	68
9.	REFERENCES	69
10.	APPENDICES	74

FOREWORD

This thesis was conducted during my employment with Aava Mobile Oy. VIOBox was an interesting project, and the software work presented countless learning experiences and long debugging sessions. I want to thank Aava Mobile VP software Janne Pulska for the opportunity to have VIOBox software as my master's thesis project. A special thanks to the head of product concepts, Mika Sipola, for all the prototype-related assistance. I would also like to extend my thanks to all the VIOBox project participants who provided me with feedback and suggestions. Many thanks also to Professor Juha Häkkinen for his guidance with the writing process. Lastly, my sincere and heartfelt thanks go to my family for all the unconditional support during my studies and this thesis project. For this, I will forever be grateful.

Oulu, June 6th, 2022

Jani Laitinen

LIST OF ABBREVIATIONS AND SYMBOLS

AES	Advanced Encryption Standard
API	Application Programming Interface
ARM	Advanced RISC Machines
CA	Certificate Authority
EHCI	Enhanced Host Controller Interface
GB	Gigabyte
GiB	Gibibyte
HCD	Host Controller Driver
HID	Human Interface Device
IP	Internet Protocol
JSON	JavaScript Object Notation
MB	Megabyte
Mbps	Megabits per second
MiB	Mebibyte
OHCI	Open Host Controller Interface
RTT	Round-Trip Time
SBC	Single-Board Computer
SCSI	Small Computer System Interface
SDK	Software Development Kit
SSH	Secure Shell
TCP	Transmission Control Protocol
TLS	Transport Layer Security
UDP	User Datagram Protocol
UHCI	Universal Host Controller Interface
URB	USB Request Block
USB	Universal Serial Bus
USB OTG	USB On-The-Go
USB/IP	USB request over IP
VHCI	Virtual Host Controller Interface
VIOBox	Virtual Input/Output Box
xHCI	Extensible Host Controller Interface

1 INTRODUCTION

The Universal Serial Bus (USB) is commonly used when transferring data between a host computer and a peripheral device. USB peripherals include storage drives, human interface devices, web cameras, and a wide variety of other devices. Additionally, special hub devices are available that increase the number of USB devices a computer can use simultaneously.

USB peripherals are usually connected to standardized ports on a computer. Traditionally, USB devices are only used locally and connected to a computer either directly or through one or more USB hubs. Due to limitations imposed by the USB standard, the maximum length of a USB 2.0 cable is approximately 5 meters [1]. There can be up to five hubs between a USB device and the host [1]. This means that even if all these hubs are externally powered, a USB device cannot be guaranteed to work from more than 30 meters away.

To address the relatively short operating distance, both commercial and open-source USB remote access software solutions have been created. Open-source software tools rely on the USB request over the Internet Protocol (USB/IP) method, which was proposed by Takahiro Hirofuchi et al. in the early 2000s [2]. USB/IP is used to transfer USB data over the Internet Protocol (IP), which allows USB devices to be theoretically used from any distance. The first open-source software tools were also created by Hirofuchi for Linux systems as a part of his research. There are also commercial USB remote sharing devices. These devices appear much like conventional hubs and can operate completely autonomously.

There is expected to be interest in a remote USB sharing solution among Aava Mobile customers. Aava currently offers two types of docking stations for use with Aava tablets. These allow charging and optionally support conventional USB hub and High-Definition Multimedia Interface (HDMI) functions. By replacing the internals of the latter data dock with the remote USB sharing hardware, it could be offered as a new accessory in the future. Simultaneously with this thesis project, a proof of concept Virtual I/O Box (VIOBox) remotely accessible USB hub was created. This prototype was built using commercial off-the-shelf components, which were evaluated to find out whether such components could be used in a commercial system.

The aim of this thesis was to document the VIOBox software design process, starting from the technical background all the way to measurements on the prototype hardware. The VIOBox software project target was to create USB sharing software tools that operate on Linux systems and implement VIOBox-specific requirements. The VIOBox device was designed to function like a regular USB hub, which meant it needed to allow clients to automatically use available devices. The system was also required to support secure data transfers between a tablet client and the VIOBox server. For cost management and customizability reasons, the software in VIOBox was to be based on open-source solutions. However, these required features were not readily available in existing open-source tools. This raised the question of: **how could open-source USB/IP software components be used to create USB sharing software tools with data security and automatic use features?**

The chapters in this thesis are organized according to the design process. In the second chapter, existing USB remote sharing solutions and design options are evaluated. This is followed by a presentation of the technical background that is required in the software work. The fundamentals of USB are presented in chapter three. Chapter four contains an in-depth look at the USB/IP method and the protocol that is used in the existing USB/IP tools. In the fifth chapter, RemoteHub, the product of the VIOBox software project, is introduced. This chapter also includes a brief overview of the mechanics and hardware of VIOBox. However, these choices are otherwise outside the scope of this thesis. The sixth chapter is dedicated to

RemoteHub software performance comparisons and other measurements. The performance of the created software is evaluated with a special focus on how the encryption performance can be optimized. Encryption is particularly expected to introduce performance degradation, and this is wanted to be known and minimized. Testing will also aim to find improvements that should be considered in the future development of VIOBox. It is especially wanted to be found: **what are the main performance bottlenecks in the prototype VIOBox system?**

2 USB SHARING OVER IP

Currently, there are many software and hardware solutions available for USB remote use. These can be useful, for example, in point of sale and warehouse use cases, and generally anywhere mobility is required. For example, a restaurant might have legacy USB receipt printers that need to be used remotely. Purchasing new devices built with remote use in mind may be more expensive than using existing USB devices with a sharing solution.

The first device for remote USB sharing was introduced by Inside Out Networks in 2001 [3]. This “AnywhereUSB™ Remote USB Over IP Concentrator” USB sharing device was initially USB 1.1 compatible and operated on a local area network. It used the Transmission Control Protocol (TCP) over an IP connection to transfer data. The maximum speed of data transfer was 12 Mbps [4].

The USB/IP method for USB remote use was proposed by Takahiro Hirofuchi et al. in the early 2000s [2]. It has also been discussed in his master’s thesis in 2004 [5] and followed up in his doctoral dissertation in 2007 [6]. As a part of his research, Hirofuchi created the first free and open-source remote USB sharing tools. These sharing tools, which are also available today, allow a Linux-based server computer to share USB devices with another Linux client computer. The USB/IP tools use a TCP/IP connection internally and support all the common USB transfer types [2]. The drivers created by Hirofuchi were added to the Linux kernel staging area in 2008 [7], followed by user space tools in 2011 [8]. The USB/IP code was merged into the mainline Linux kernel in 2014, starting with kernel version 3.17 [9].

Today, both commercial and open-source USB sharing tools have added many new features on top of those initially implemented. These include features such as USB 3.0 [10] and Windows support [11] in the open-source community. Commercial tools have further implemented data security, automatic server discovery, fine-grained access controls, and a broad range of other features. Although development has been active in the open-source community, many of the commercial tool features are yet to be included in open-source tools.

This chapter presents background information about the currently available USB sharing devices and software. A special focus is on the capabilities of the previously mentioned USB/IP tools in the Linux kernel and the subsequently developed Windows port of these tools.

2.1 Commercial devices

Multiple commercial vendors offer remote USB sharing hubs. These devices bundle the hardware and software in one package. The most notable is the AnywhereUSB™ line of devices. The current AnywhereUSB™ plus devices are available with two to twenty-four USB 3.1 ports. These hubs are managed using a web interface or through a built-in command line with Secure Shell (SSH) or serial connection. The AnywhereUSB™ hubs also support data encryption, can be automatically discovered, and there is extensive support for access control. [12] For example, Solid State Supplies Ltd offered the two-port variant for 263€ excluding taxes [13].

There are also similar devices from other vendors, such as AnyplaceUSB hubs from Coolgear Inc. These AnyplaceUSB devices support network traffic encryption, password authorization, and automatic discovery. They support Linux and Windows operating systems, but only contain USB 2.0-capable ports. [14] At the time of writing, USBGear.com was selling the two-port device for 61€ [15].

2.2 Commercial software

Remote USB sharing software can also be purchased separately if a user opts to use their own server hardware. One well-known commercial tool is VirtualHere, which provides a USB device sharing server application for Linux, Windows, macOS, Android, and specific Network Attached Storage (NAS) devices. The VirtualHere client software is available for Linux, Windows, macOS, and Android devices. VirtualHere includes support for data security and compression. The client application supports finding servers automatically on a local network. Also, server discovery on public network is supported using the EasyFind service [16]. VirtualHere can be customized with the use of configuration files and scripts. The system can use, for example, IP, password, or USB device properties to grant or deny access to a particular device. Overall, VirtualHere is a feature-rich USB sharing solution. At the time of writing, it cost 46€, which included a license for one server. The number of clients for a server was unrestricted. In addition, there is an active support forum on the VirtualHere website where the founder answers questions and resolves problems that users may have. [17]

Another software example is “USB Network Gate” by Electronic Team. The USB Network Gate supports Windows, Linux, Android, and macOS systems and allows USB communication to be secured and compressed. The client application can also find servers automatically on a local network. The USB Network Gate is available for use in multiple ways. In addition to conventional server and client applications, the service can be purchased as a Software Development Kit (SDK). By using the SDK, implementers can add the USB sharing capability to their services or apps. The full application source files are also available for licensing. The price for commercial use, the SDK, and source licensing is determined on a case-by-case basis by contacting Electronic Team. The applications were available for personal and non-commercial use for 149€. [18]

2.3 Open-source USB/IP software

USB/IP is a method for transferring raw USB data over an IP network using a virtual host controller on the client computer and a special device driver on the server computer. From the client computer’s point of view, the remote USB device appears just as if it were physically connected to it [2]. The technical details of both USB and USB/IP are studied in later chapters. Because the software developed in this work is based on open-source USB/IP methods and software components, these are examined in greater detail. This section presents the features and user interface of existing USB/IP tools.

2.3.1 Linux tools

The USB/IP tools are used from the command-line. With the command line interface, users can find, attach, and detach remote USB devices. Only the user interface and functions are explored in this section. The USB/IP protocol and drivers are explored in-depth in chapter four.

The Linux USB/IP user space tools are comprised of *usbip* and *usbipd* applications. The *usbip* executable is used by both servers and clients. On the server, it is used to bind USB devices to be used with USB/IP. After binding, devices are allowed to be used by clients and cannot be used locally by the server. Clients use the *usbip* executable to query servers for

device listings and attach devices that are bound for use. A successful attach with the tool is equivalent to plugging a USB device into the client system.

Servers use the *usbipd* application. It is a daemon process that receives commands from *usbip* clients and starts USB forwarding when possible. The *usbipd* application is designed to be used in the background but can print debugging information about its internal operation.

The user space applications control the USB/IP kernel drivers. There is a Virtual Host Controller Interface (VHCI) Host Controller Driver (HCD) *vhci-hcd* Linux kernel driver that is used on the client. It implements a virtual host controller with a virtual root hub for transferring USB data and is referred to as the VHCI driver in this paper. Servers use a “STUB” *usbip-host* Linux kernel device driver module that allows raw data transfers with USB devices.

The following Figure 1 illustrates the steps that a server needs to execute to enable a USB device to be used by clients. The steps that are run using *sudo* indicate they need root privileges.

1. The USB device is physically attached to the server computer and the *usbip-host* module is loaded. Internally, the *modprobe* tool loads also a required *usbip-core* module. This command needs to be executed only once.
2. The *usbipd* process is started if it is not already running. The “&” at the end of the command indicates that *usbipd* is started in the background. However, because the output is not redirected, informative traces are printed on the terminal. The process can also be started as a daemon process by supplying “-D” parameter during startup.
3. A local device list is queried with *usbip* to find out the “busid”, which is a sequence of USB bus and port numbers where the USB device is physically attached.
4. The USB device is marked as exportable by binding it. The unique port configuration found during step 3 is used to refer to the device.

```
[debug@debug-computer: sudo modprobe usbip-host
[debug@debug-computer: sudo usbipd &
[1] 64806
debug@debug-computer: usbipd: info: starting usbipd (usbip-utils 2.0)
usbipd: info: listening on 0.0.0.0:3240
usbipd: info: listening on :::3240

[debug@debug-computer: usbip list -l
- busid 1-1.3 (21b4:0083)
  unknown vendor : unknown product (21b4:0083)

- busid 1-1.4 (046d:c051)
  Logitech, Inc. : G3 (MX518) Optical Mouse (046d:c051)

- busid 2-1.2.1 (413c:3012)
  Dell Computer Corp. : Optical Wheel Mouse (413c:3012)

- busid 2-1.2.2 (0951:1666)
  Kingston Technology : DataTraveler 100 G3/G4/SE9 G2 (0951:1666)

[debug@debug-computer: sudo usbip bind -b 2-1.2.1
usbip: info: bind device on busid 2-1.2.1: complete
```

Figure 1. Steps to start the USB/IP server and allow a Dell mouse that is physically attached to the server to be used by clients.

After the server has bound a USB device for use, clients can start using it. The steps for a client to attach a remote USB device are shown in Figure 2 and explained below.

1. The *vhci-hcd* driver module is loaded if not already present. Again, the *usbip-core* module is also automatically loaded.
2. A device listing command is issued to the IP address of the server. The resulting listing shows all bound but still unattached devices on the server.
3. An attach command is issued to start using the device. This command contains the unique port configuration of the device on the server.

```
debug@debug-computer: sudo modprobe vhci-hcd
debug@debug-computer: usbip list -r 192.168.1.185
Exportable USB devices
=====
- 192.168.1.185
  2-1.2.1: Dell Computer Corp. : Optical Wheel Mouse (413c:3012)
           : /sys/devices/pci0000:00/0000:00:1d.0/usb2/2-1/2-1.2/2-1.2.1
           : (Defined at Interface level) (00/00/00)
           : 0 - Human Interface Device / Boot Interface Subclass / Mouse (03/01/02)
debug@debug-computer: sudo usbip attach -b 2-1.2.1 -r 192.168.1.185
```

Figure 2. Steps for a client to start using the Dell mouse that was previously bound on the server.

After the previously illustrated negotiation in user space programs has successfully finished, the open network sockets are passed to the USB/IP kernel drivers by both client and server applications. The kernel space *usbip-host* STUB device driver on the server and virtual host controller driver *vhci-hcd* on the client then communicate with each other, transferring USB data using the protocol defined by the drivers. This kernel space USB/IP tool protocol is presented later in detail. User space applications are not involved in the transfer of USB data and communicate with each other using the user space USB/IP tool protocol that is also presented later.

2.3.2 Windows tools

The USB/IP Windows utilities were first introduced in 2009 by SourceForge user “leptonwu” [11]. The Windows tools and drivers have been actively developed ever since, and currently the main software project is USBIP-win [19]. This Windows software suite is compatible with the Linux USB/IP tools. Like its Linux counterpart, it provides user space server and client applications that implement a command-line interface along with kernel drivers. Commands for Linux *usbip* generally work in the Windows version as well. Available commands are shown in Figure 3.

```

PS C:\Users\debug\Documents\usbip-win-master\Release\x64> .\usbip.exe
usage: usbip [--debug] [--tcp-port PORT] [version]
      [help] <command> <args>

  attach      Attach a remote USB device(WDM or ude)
  detach      Detach a remote USB device
  list        List exportable or local USB devices
  bind        Bind device to usbip stub driver
  unbind      Unbind device from usbip stub driver
  install     Install usbip vhci driver
  uninstall   Uninstall usbip vhci driver
  port        Show imported USB devices

```

Figure 3. Commands available in the Windows USB/IP tool usbip.exe.

One notable disadvantage of the Windows tools is that, at the time of writing, the client virtual host controller driver was not digitally signed. This means that the driver can only be installed in Windows test mode, in which driver signatures are not enforced. This is a problem in a commercial setting because end users cannot be expected to use their computers in test mode. It is possible to sign the driver by purchasing a specific signing certificate and submitting the driver to the Microsoft developer portal [20]. This process can be costly, and getting the drivers signed may be difficult for open-source projects; it is more geared for organizations.

2.3.3 *Limitations and possible solutions*

The current USB/IP tools for Linux and Windows lack features such as data security and automatic discovery built in. Methods to overcome the key limitations of current USB/IP tools are explored in this section. The areas of interest include data security, automatic discovery, and unattended use. Other limitations are not explored here, such as the lack of fine-grained access controls, which may be important depending on the use case. Hirofuchi presented in his doctoral dissertation an implementation of a prototype sharing system with data security and automatic use [6]. However, these features are not implemented in the current public tools.

Security

One widely used framework for assuring information security is the CIA triad [21]. The CIA triad, originally introduced in the 1970s, refers to three core security concerns in information systems. These are Confidentiality, Integrity, and Availability. Confidentiality is needed so that information cannot be read by unauthorized actors. It can be implemented with the help of various encryption algorithms, such as the Advanced Encryption Standard (AES) and the Rivest–Shamir–Adleman (RSA) cryptosystem. Integrity is required so that data cannot be tampered with by unauthorized actors, leading to legitimate users unknowingly using invalid data. Integrity would be ensured, for example, by using hash functions and cyclic redundancy check (CRC) algorithms. Availability refers to the service being available at any time so that third parties cannot prevent authorized users from accessing it. Availability aspects would include, for example, Distributed Denial of Service (DDoS) mitigation.

Confidentiality and integrity aspects can be implemented, for example, by using Transport Layer Security (TLS) or SSH protocols. TLS is a very common protocol that is used for encrypting network transfers. Many versions of the TLS and its predecessor Secure Sockets Layer (SSL) have been introduced, the latest of which is TLS 1.3. A TLS connection is initialized with a handshake, usually using certificates, which are used to verify the authenticity of connection participants and establish later data encryption. The initial handshake is, in most cases, performed with asymmetric encryption and public-key cryptography, where connection participants can only either encrypt or decrypt data. After the handshake, symmetric keys are used for later data transfers that allow both encryption and decryption. Trust is usually established by using a trusted third party referred to as a Certificate Authority (CA) [22]. The CA signs server certificates with its confidential private key. This allows a client to verify whether the certificate was signed by a CA by using the CA's public key. A successful verification enables a client to trust the certificate and its metadata. The CA certificates can also be self-signed, which means no trusted third party is mandatory. The following Figure 4 shows a simplified TLS 1.2 handshake process that assumes a Diffie-Hellman-based key exchange and RSA public-key authentication [23][24].

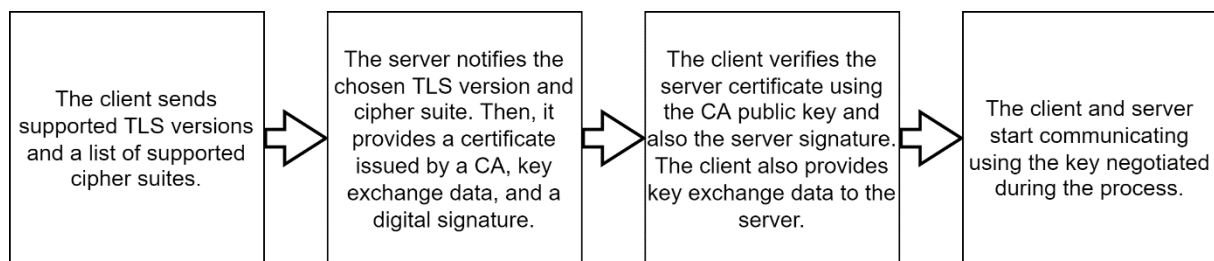


Figure 4. Simplified process of a TLS 1.2 handshake.

TLS requires a reliable transport method to carry the protocol. TLS often uses TCP when transferring data over a network and relies on it for possible packet retransmissions and packet ordering. TLS 1.2 implementations usually support many cipher suites that define the algorithms for different tasks that are authentication, encryption, and integrity verification [25]. As shown in Figure 5, TLS 1.2 cipher suites are represented in a unified way. However, TLS 1.3 cipher suite definitions are different to the previous versions. These cipher suite components define only the symmetric ciphers, and the key exchange and authentication ciphers are not included.

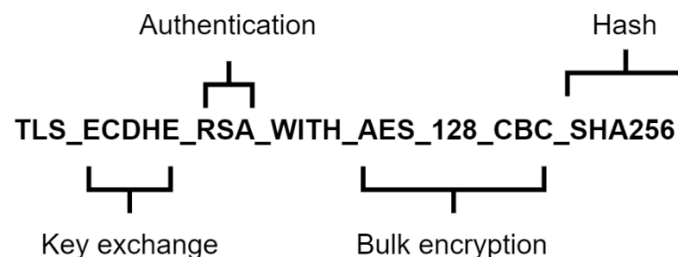


Figure 5. An example TLS 1.2 cipher suite with a breakdown of the components.

Ciphers used in each of the tasks can be read from the cipher suite description and have different performance and security implications. In particular, the chosen encryption and

integrity ciphers can have a great impact on performance, in addition to the apparent security aspects.

Another option for introducing confidentiality and integrity is to use SSH port forwarding, or “tunneling”, as it is commonly called. By tunneling data with SSH, the current USB/IP tools do not need any modifications, and all the user space and kernel space data transfers can be secured with minimal effort. It should be noted that when using USB/IP with SSH tunneling, it is necessary to deny access to the USB/IP service port to enforce the use of the tunnel. Depending on implementation, SSH tunneling may support data compression that could increase data throughput.

Takahiro Hirofuchi stated in his doctoral dissertation that Internet Protocol Security (IPsec) would be suitable for USB/IP [6]. In his tests, the performance of IPsec would be approximately 50 % compared to an unencrypted configuration, depending on the used encryption algorithm. There are clues in the source code of USB/IP where IPsec support could be added, but the current USB/IP tools on Linux and Windows do not implement it.

Ensuring availability is also an important aspect that needs special attention, especially when operating in the public network environment. Tools have been created to ensure availability, for example by banning clients from accessing a service after a predefined number of unsuccessful attempts. In the case of existing USB/IP, a router-level firewall must not be opened to allow access from outside of the local network since USB/IP does not support any authentication. Even with authentication, a firewall is still useful for preventing malicious actions such as denial of service attacks from outside of the local network.

Automatic discovery

Automatic discovery refers to the capability of a client to find servers automatically without being aware of server IP addresses. This is vital so that the system is as easy to use as possible. Automatic service discovery is commonly implemented with multicast Domain Name System (mDNS) tools. In Windows, there is an Apple-developed software package that implements this automatic service discovery called Bonjour, and in Linux, there is an implementation named Avahi. There are libraries and command-line tools available on Linux that enable finding and publishing services, and a Bonjour SDK for adding automatic discovery to Windows applications.

It is also possible to create a custom lightweight software solution that uses User Datagram Protocol (UDP) broadcast transfers. The transferred packets should include information such as the port where clients can find the server. Servers would need to send these packets to a special broadcast address, which causes a router to distribute them to all clients on the same network. Clients can then use information encoded in the packets to automatically establish a connection.

Automatic use

For ease of use, USB/IP tools should be automated so that they do not require any user interaction. The first low-effort method would be to use scripts on server and client computers. The scripts would allow associated software components to be controlled from a single location. They can also be started automatically when a device is powered on.

The script for the server would need to do the following:

- Load the USB/IP server drivers
- Start the *usbipd* daemon process
- Publish server name and broadcast availability for clients using Avahi
- Bind USB devices automatically for use with USB/IP when they are plugged into the system

The script for the client needs to do the following:

- Load the USB/IP client drivers
- Monitor available servers using Avahi
- Initialize SSH connections with servers
- Send device listing queries and attach all devices from identified servers

It is possible to create such scripts for Linux and Windows. They should be implemented as services so that users do not need to interface with them. However, although the usage of these scripts would be sufficient for casual users, they are not well suited for commercial applications. The scripts require multiple separate applications to function that would also need to be included on the target systems. It would be desirable to implement all the logic in standalone executables. Especially when the client may need to run on Windows and Android in the future.

3 INTRODUCTION TO USB

The previous chapter introduced the features, strengths, and limitations of both open-source and proprietary remote USB sharing tools. This chapter presents the software aspect of USB generally and in the context of the Linux kernel. This helps understand the internal operation of existing USB/IP tools and, subsequently, the software work that was done during this thesis.

3.1 Overview

USB is widely used in personal computers and a broad range of consumer appliances in general. The first USB standard version 1.0 was introduced in 1996. Initially, there were only two data rates available, which were low-speed and full-speed, with theoretical maximum speeds of 1.5 Mbps and 12 Mbps, respectively [1]. The theoretical maximum speeds of different USB standards are shown in Table 1. In practice, the actual useful payload transfer speed will be lower for a variety of reasons, including USB protocol overhead and transfer type characteristics.

To respond to the increasing performance and usability demands, the USB standard has been constantly evolved and updated. The updates have been implemented in a backward-compatible manner. Users have been able to keep using older devices when new versions of the USB standards have been introduced. Hardware vendors can also take advantage of older and slower USB versions to reduce the manufacturing costs of devices that do not require state-of-the-art speeds.

Table 1. The theoretical maximum speeds of different USB standard versions

USB standard	Maximum speed
1.0 / 1.1	1.5 Mbps Low-Speed and 12 Mbps Full-Speed
2.0	480 Mbps High-Speed
3.0 (aka. 3.1 Gen1 and 3.2 Gen1)	5 Gbps SuperSpeed
3.1 (aka. 3.1 Gen2 and 3.2 Gen2)	10 Gbps SuperSpeed+
3.2	20 Gbps (Maximum speed with Type-C only)
4	40 Gbps (Type-C only)

USB was initially intended to only connect a host computer to a peripheral. The data of early USB devices was only transferred over a cable that was terminated with A-type and B-type plugs. The A-type plug connects to a receptacle on the host computer, and likewise, the B-type plug to a receptacle on the peripheral. The plugs were quite large and not suitable for portable devices. They were later introduced in smaller mini and micro sizes, which have found use in a broad range of devices. A Type-C connector was introduced in 2014, which is rotationally symmetrical and allows both peripheral and host to use the same type of plug.

USB versions 1.0, 1.1, and 2.0 use four pins in the connectors, which are: one ground pin, one power pin, and two data pins for half-duplex communication. USB 3.0 data is full-duplex and requires four pins for data. The USB 3.0 and later connectors also include the legacy data connectors for backward compatibility. In Figure 6, the USB A-type plugs are shown.

The USB data is transferred over a USB cable with the help of differential signaling. This means the same data is transferred on two data lines with opposite polarities. Because of this, noise can be subtracted out of the signal if the same error is present on both lines.

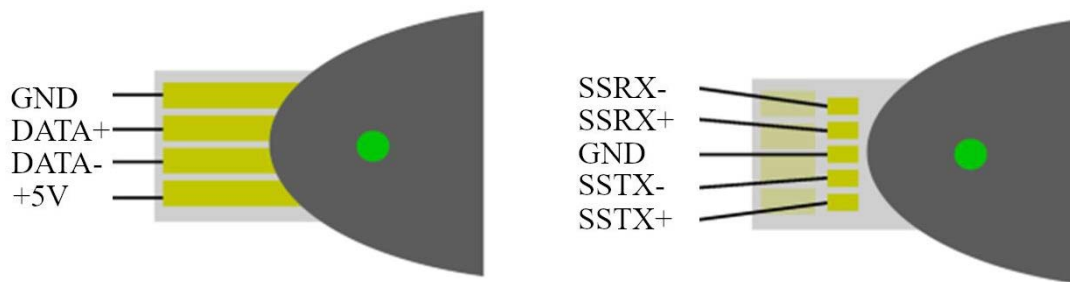


Figure 6. USB A-type 1.0/1.1/2.0 plug includes 4 pins (on the left), and USB 3.0 adds 5 more connectors to the A plug (on the right).

3.2 Data

Data transfers to USB devices are initiated by the host computer. The host communicates with a USB peripheral by accessing endpoints, which are buffers that can either accept or provide data [26]. There can be up to 32 endpoints in a peripheral. At least two endpoints are present in every USB device and are reserved for control. The logical connection between the system software and an endpoint is referred to as a “pipe”. The previously mentioned control endpoints are accessed with the default control pipe that allows both reading and writing. The default control pipe is a message pipe accessed using USB-defined control transfers. All data in a message pipe must have a USB-mandated format as defined in the USB specifications. Other pipes target numbered endpoints and are used for the actual useful functions of a USB device. These pipes are stream pipes and use interrupt, bulk, and isochronous transfer types. The data structure in stream pipes is not defined in the USB specifications. Therefore, data in stream pipes can contain higher-level protocols that are only carried over USB. [1]

When a device is attached to a host computer, the host first learns information about the device and configures it for use in a process called enumeration [27]. During enumeration, the host uses the default control pipe to issue command requests. With these requests, the host assigns a unique address, reads information descriptors, and in general configures the device for use. The device should be ready for its intended function after enumeration and have one or more drivers assigned to it. The information in this section contains a high-level overview of the common USB functions, which in general apply to all USB standards. The information contained in this chapter summarizes the main software features presented in the USB 2.0 specification.

3.2.1 Requests

A host uses USB requests to control and learn information about a USB device. Requests are directed to the default control pipe and follow a standard format that describes an action to be performed by the device. There are many request types available. Some are standard requests that must be implemented in every USB device. Additionally, there are class and vendor-specific requests. The standard requests notably include `GET_DESCRIPTOR` and `SET_ADDRESS`, which are among the first requests a host issues during enumeration. The default control pipe can serve standard requests at any time, regardless of whether enumeration has been completed.

3.2.2 Descriptors

Descriptors are read from a USB device with a GET_DESCRIPTOR request. They contain essential information about a peripheral and can be standard-, class-, or vendor-specific. Every USB device supports standard descriptors, and a host will always use these to gain information during enumeration. Standard descriptors include device, configuration, interface, and endpoint descriptors. These descriptors are stored in a tree hierarchy where the higher-level descriptor informs the presence of lower-level descriptors. There may be additional standard descriptors supported depending on the USB standard, but they are not in the scope of this chapter.

The first descriptor a host reads from a device during enumeration is the device descriptor that contains the most high-level overview of the device. The device descriptor includes, for example, vendor and device-specific identification numbers, the number of supported configurations, and how to read human-readable names contained in string descriptors. There can only be one device descriptor in a USB device.

Configurations can change the function of the device fundamentally, and they are exposed using configuration descriptors. There can be only one configuration active at a time. For example, a USB dongle could show up either as storage or as a modem if these functions are defined in separate configurations. The configuration descriptor informs about the maximum power draw after enumeration, among other things. Although possible, it is uncommon for a device to have more than one configuration [28]. The host finds the number of interface descriptors within the configuration descriptor and parses them next.

Interface descriptors contain information about the interfaces and alternate settings. Interfaces define the logical functions of a USB device. There can be many active interfaces after enumeration. The host computer attempts to assign a driver for each interface, and in this way, a single USB device can provide multiple separate functions. Devices that have multiple independently controlled interfaces but only a single address are called composite devices. Interfaces can support multiple alternate settings for controlling endpoints slightly differently after configuration. As an example, a web camera might output video in different resolutions depending on the alternate setting. Each interface has one or more endpoints, the number of which is contained in the descriptors.

The endpoints are exposed using endpoint descriptors, which inform the data transfer direction, maximum packet size, and transfer type of the transactions to endpoints. The host uses this information later when communicating with the device using the endpoint.

The Figure 7 illustrates the descriptors as read from a HP optical mouse. This represents a simple descriptor hierarchy. It contains standard descriptors and a class-specific Human Interface Device (HID) descriptor, which informs of the presence of an HID report descriptor. Control endpoints are assumed to always be present and do not have endpoint descriptors.

Bus 002 Device 006: ID 03f0:134a HP, Inc Optical Mouse

Device descriptor:	
bLength	18
bDescriptorType	1
bcdUSB	2.00
bDeviceClass	0
bDeviceSubClass	0
bDeviceProtocol	0
bMaxPacketSize0	8
idVendor	0x03f0 HP, Inc
idProduct	0x134a Optical Mouse
bcdDevice	1.00
iManufacturer	1 PixArt
iProduct	2 HP USB Optical Mouse
iSerial	0
bNumConfigurations	1

Configuration Descriptor:	
bLength	9
bDescriptorType	2
wTotalLength	0x0022
bNumInterfaces	1
bConfigurationValue	1
iConfiguration	0
bmAttributes	0xa0 (Bus Powered) (Remote Wakeup)
bMaxPower	50 (100mA)

Interface Descriptor:	
bLength	9
bDescriptorType	4
bInterfaceNumber	0
bAlternateSetting	0
bNumEndpoints	1
bInterfaceClass	3 Human Interface Device
bInterfaceSubClass	1 Boot Interface Subclass
bInterfaceProtocol	2 Mouse
iInterface	0

HID Device Descriptor:	
bLength	9
bDescriptorType	33
bcdHID	1.11
bCountryCode	0 Not supported
bNumDescriptors	1
bDescriptorType	34 Report
wDescriptorLength	46

Endpoint Descriptor:	
bLength	7
bDescriptorType	5
bEndpointAddress	0x81 EP 1 IN
bmAttributes	3 (Transfer Type: Interrupt) (Synch Type: None) (Usage Type: Data)
wMaxPacketSize	0x0004 1x 4 bytes
bInterval	10

Figure 7. Illustration of USB descriptors in a HP Mouse obtained with *lsusb -v* command and string descriptors have been parsed.

3.2.3 Transfer types

The USB data transfer types determine what transfer characteristics are favored, such as bandwidth or latency [29], and they are used to support the different requirements of USB devices. The transfer type of an endpoint is read from endpoint descriptors. USB data can be transferred with control, interrupt, bulk, or isochronous transfer methods. The following is a short introduction of each of the transfer types.

Control

Every USB device supports control transfers on the default pipe. They are delivered with a “best-effort” strategy. Control transfers are used for querying device-specific information and configuring the device. For instance, the requests and descriptors are handled using control transfers. The control transfers use a message pipe, which means that the format of the data has a USB defined format.

Bulk

Bulk transfers are allocated when the bus is not busy with other transfer types. Bulk transfers ensure delivery without guaranteed bandwidth or latency. This transfer type is commonly used in flash drives, which transfer large amounts of data.

Interrupt

Despite the name, interrupt transfers are host-initiated. They are suitable for transferring data periodically. For example, keyboards, mice, and other human interface devices commonly use interrupt transfers. These devices may only have new data available at regular intervals.

Isochronous

Isochronous transfers are scheduled at fixed intervals, allowing a constant data rate. Transfers are not retried in the case of delivery failure. Isochronous transfers are often used with real-time audio and video devices.

3.3 Hardware

This section gives an overview of the components for host USB support. This view does not include USB On-The-Go (USB OTG) support, where a device can act as a host or a peripheral.

3.3.1 Host controller

The host controller is conventionally a physical hardware module that enables communication with USB peripherals. Physical USB wires connect to the host controller and data is driven to the bus with the help of a root hub contained in it. Host controllers use physical layer circuitry to implement the low-level physical aspects of USB required for communicating with peripherals. The host controller may be a separate chip or included as an intellectual property block implemented in custom silicon. The host controller itself can connect to the computer in various ways. Commonly, in desktop computers, the host controller connects via the Peripheral Component Interconnect (PCI) bus, but many other options are available.

There are four commonly used interface standards for physical host controllers: Universal Host Controller Interface (UHCI) and Open Host Controller Interface (OHCI) for USB 1.0

and 1.1; Enhanced Host Controller Interface (EHCI) for USB 2.0; and eXtensible Host Controller Interface (xHCI), which is designed to replace the previous host controller standards and supports USB speeds including USB 3.0. These interfaces define the methods for communicating with the host controller. Each interface requires its own set of host controller drivers.

3.3.2 *Hub*

USB hubs have the special purpose of extending the number of USB ports on a computer. Hubs appear as regular USB devices that implement hub class functionality. They can be stand-alone devices but are sometimes integrated into peripherals. Host controllers contain a root hub integrated into them that is the first device on a USB bus [28]. The root hub should appear and function as any other hub.

USB hubs are polled continuously to detect attached or removed devices. The hubs have an interrupt endpoint which, when polled, reports the status changes of each of the ports. After the host learns about a change, it can then take appropriate action. Some of the responsibilities of hubs include suspending devices and disabling ports as requested by the host. High-speed capable hubs contain a transaction translator that converts high-speed transactions to lower-speed transactions for devices that require it.

3.4 Software in the Linux kernel

This chapter has so far offered a general introduction to USB hardware and software. The USB 2.0 specification suggests how software for USB support should be implemented. The specification breaks the USB support into layers, which all have distinct tasks. In a simple view, these layers consist of client software for using the functions defined by USB device interfaces; the USB driver, which abstracts host controller implementations; and host controller drivers, which communicate with host controller hardware. USB data is moved between the client and the host controller drivers in I/O Request Packets (IRPs). This layered structure is also present in the Linux USB stack.

This section attempts to illustrate how the USB functions are implemented in the Linux kernel. Depending on the hardware, computers may support USB OTG, which allows both host mode and peripheral mode. The peripheral mode is used by many mobile devices to present themselves as a USB storage device or a serial communication device. This dual role mode of operation is not explored in this chapter.

3.4.1 *USB Request Block (URB)*

The USB Request Block (URB) is a key data structure in the Linux USB subsystem. An URB is used when transferring data to or from a USB device. It carries USB transfers between the layers of the Linux USB stack. The URB contains all the data necessary to complete a USB transfer and a callback function that is called after the transaction has been completed. Any USB transfer can be expressed with a URB. The URB also includes a buffer for holding the USB data and pipe information.

The URB contains many fields [28], but due to this thesis focusing on USB/IP, Table 2 lists only the URB structure public member variables that are transferred with the USB/IP

protocol. A full list of data fields in URB can be found in the kernel source tree header file `/include/linux/usb.h`.

Table 2. The USB/IP relevant fields in a Linux URB

Variable name	Description
status	Status for a completed URB. Valid for transfer types other than isochronous.
transfer_flags	Transfer flags modify the handling of a URB. For example, should reading a shorter amount of data than requested be treated as an error.
transfer_buffer	The buffer that holds the data that is written to a USB device or filled with data from a USB device.
transfer_buffer_length	The size in bytes of the transfer_buffer. It is the amount of data that is requested to be read or written.
actual_length	The number of data bytes that were actually written or read.
setup_packet	Eight bytes that are always used in control transfers (bmRequestType, bRequest, wValue, wIndex, and wLength).
start_frame	Initial frame number for isochronous transfers.
number_of_packets	The number of isochronous transfer packets contained in the transfer buffer. The description of which is contained in iso_frame_desc.
interval	Polling interval of interrupt and isochronous transfers.
error_count	The number of failed isochronous transfers.
iso_frame_desc	Information about the isochronous buffers in transfer_buffer. This allows a single URB to define multiple isochronous transfers.

3.4.2 Linux USB stack

The USB support in the Linux kernel is implemented in a layered manner. In the simplest view, the USB stack consists of device drivers, the USB core, and host controller drivers. Device drivers, in general, implement the functions of USB devices. The USB core abstracts host controller differences from USB device drivers. Host controller drivers access host controllers in a hardware-defined manner. Figure 8 presents a simplified overview of the Linux USB stack.

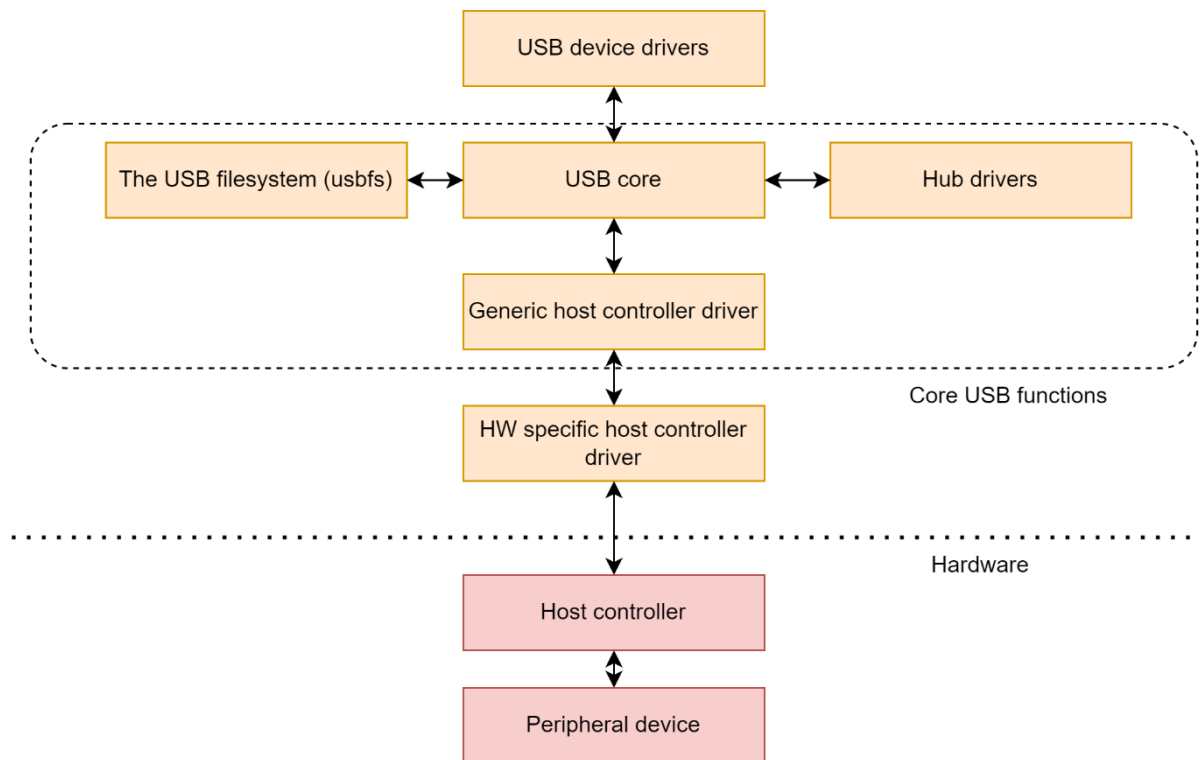


Figure 8. A simplified high-level illustration of Linux USB support.

Device drivers

USB device drivers bind to interfaces [28], which allows one USB device to provide multiple functions. On top of the USB device driver layer, there are other Linux subsystems [28] that decouple the USB transport method from the actual useful functions. Therefore, a device that implements a specific software interface could be attached to the system using many different hardware interfaces. The carried protocols might include, for example, Small Computer System Interface (SCSI) commands for flash drives and the HID protocol for input. In essence, the USB itself is a data channel that requires no structure for the data it delivers [28]. USB device drivers can be class-specific or vendor-specific. A vendor-specific driver is required when a device implements a custom software interface. However, if a device conforms to a predefined class, a hardware vendor does not need to provide custom drivers since most classes are supported in the Linux kernel [30].

There is no single place for USB device drivers in the Linux kernel. For example, USB mass storage drivers can be found in *drivers/usb/storage*. These USB mass storage drivers submit URBs and provide a layer for communicating with upper-level SCSI drivers using the SCSI protocol [31]. The logic of USB SCSI mass storage is thus handled by the Linux SCSI subsystem. Device drivers use functions defined in the USB core to transfer data using URBs. This interface is exposed in *include/linux/usb.h*. Device drivers can use *usb_control_msg()*, *usb_submit_urb()*, *usb_get_descriptor()*, *usb_set_interface()* and others.

The USB core and core functions

The USB core sits between device drivers and host controller drivers [28] and can be essentially thought of as an abstraction layer for device drivers to access host controllers [31]. The USB core can support multiple different USB device drivers and host controllers. Other core capabilities, such as a USB filesystem (*usbfs*) and a hub device driver, are closely integrated in the core to provide USB support. The USB core, along with other core functions, can be found in the folder *drivers/usb/core* in the Linux source distribution.

The USB core provides interfaces for device drivers to access a USB device without knowledge of the used host controller implementation. There are two Application Programming Interfaces (APIs), one for general-purpose drivers and the other for essential drivers that are part of the core, such as a hub driver and HCDs [32]. As an example, the USB core provides as a part of the device driver interface *usb_submit_urb()* and *usb_kill_urb()* functions that queue and cancel USB I/O requests, respectively. When a USB device is first inserted into the system, the hub driver detects this. After initial detection by the hub driver, the device is enumerated, which should bring the device to a functional state.

The *usbfs* provides support for developing user-space USB drivers [31]. This allows devices to be detached from enumeration-time assigned drivers and provide full control of USB data transfers into user space. There is a libUSB library available that wraps this interface into an easier-to-use form [33].

Host controller drivers

A host controller driver accesses the underlying host controller hardware through a hardware-specific interface. These drivers are managed by the USB core. The core USB functions in Linux include a host controller driver framework, which delegates to a hardware-specific driver only when necessary [34]. Most of the host controller driver implementations are located at *drivers/usb/host* in the kernel source tree. That folder includes drivers for previously introduced OHCI, UHCI, EHCI, and xHCI host controller implementations. The USB/IP VHCI host controller drivers are in the Linux kernel source tree in folder *drivers/usb/usbip*.

4 USB/IP COMPONENTS AND PROTOCOL

USB request over IP (USB/IP) is an operating system independent way of extending USB over an IP network [6]. It was introduced by Takahiro Hirofuchi et al. and operates by using a virtual host controller on the client and a universal device driver on the server capable of communicating with any USB device. USB/IP allows USB devices to be used remotely just as if they were connected locally. This chapter summarizes and presents information about the existing implementation of USB/IP on Linux systems and presents the existing USB/IP tool protocol. Initially, USB/IP was only available on Linux, but has since then been extended to work on Windows as well.

In the previous chapter, it was found that USB software support in Linux consists of three fundamental layers: device drivers, USB core, and host controller drivers. The host controller driver conventionally controls a physical host controller device that communicates with USB devices. Requests for data transfers usually originate from enumeration time assigned device drivers, which operate a device using the protocol implemented by the device. USB/IP on a client replaces the host controller layer with a virtual implementation that wraps URBs it receives with a USB/IP specific protocol and transfers them to the server using a TCP/IP connection. This VHCI driver is operated by the USB core using the same host controller interfaces as physical controllers. At the server end, a STUB device driver receives the URBs. The STUB driver is a universal device driver that allows any type of USB data transfer to be performed as instructed by client-side device drivers. The STUB completes the received URBs through a regular USB stack on the server. This method joins the two USB stacks on the client and server computers, providing the client access to server devices as if they were locally connected. Figure 9 illustrates these layers [6].

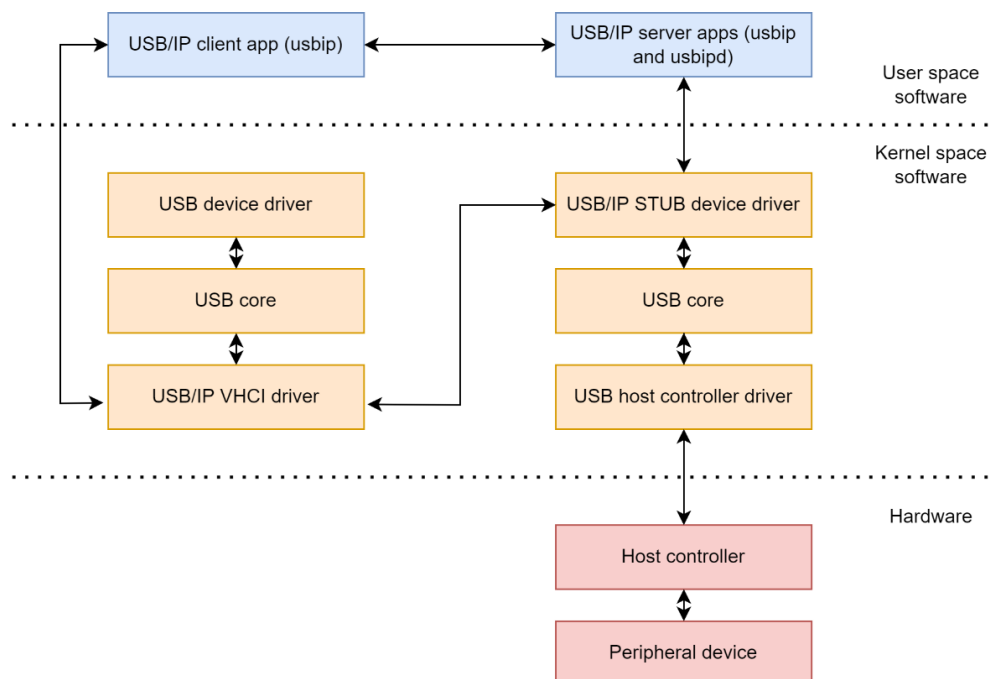


Figure 9. An illustration of the hierarchy of USB and USB/IP module interactions on two separate computers (client on the left and server on the right) connected by a TCP/IP connection.

The USB/IP tools include user space applications for controlling the USB/IP system, which is otherwise mostly implemented in the kernel. These applications were explored in the second chapter. The applications control the USB/IP kernel drivers via the *sysfs* interface. The *sysfs* is a pseudo filesystem that is usually mounted in the */sys* folder. The logic of determining which devices are to be used is performed in user space, and USB data is transferred in kernel space. User space applications and kernel space drivers talk with each other using distinct protocols. Both protocols [35] are explored in detail in this chapter.

4.1 User space applications

The USB/IP user space tools consist of two executables, which are *usbip* for both server and client, and *usbipd*, which is used on the server only. The core function of the user space applications is to allow a host to find a suitable device for use and then facilitate the start of USB data transfers. To achieve this, the USB/IP system supports device listing and attaching commands in the user space USB/IP tool protocol.

Before USB devices can be used by clients, they need to first be bound on the server to the STUB driver, after which they will be attachable by clients and present in device listings. Then, after the user space tools have negotiated a USB device to be exported, they pass open network sockets to the kernel drivers where USB data transfer takes place. This means no USB data is transferred between *usbip* and *usbipd* executables, and the USB requests are transferred between the kernel drivers.

Linux USB/IP tools also include virtual USB device controller functionality. It is implemented in a *usbip-vudc* driver that is loaded on the server. This allows the host to export gadget devices (emulated peripherals) to remote clients. This mode of function is not explored in this chapter. User space USB/IP-related source code can be found in the kernel source tree folder location *tools/usb/usbip*.

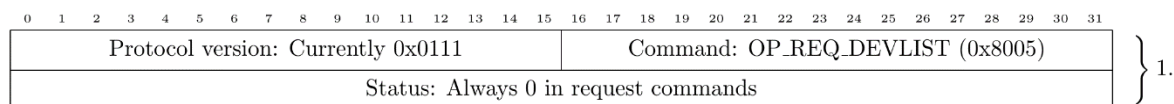
4.1.2 Protocol and capabilities

The *usbip* and *usbipd* processes communicate with the user space USB/IP tool protocol. The protocol contains, at the time of writing, two Protocol Data Units (PDUs), which are referred to as commands in this chapter. One command is for remote device listing and the other is for attaching. Both commands originate exclusively from clients. The rest of the application functions need no network communication.

At the protocol level, all USB/IP user-space commands include an 8-byte long header that is present in every transaction. The header contains a version number for detecting protocol support, a command type, and a status field that is used for checking that the command was executed successfully on the server. Packets sent from the client to the server are named with “REQ” prefixes, indicating they request an action to be performed by the server. Server replies are named with “REP” prefixes, indicating they contain a reply to a request. The following sections present the user space protocol and application-supported features in detail.

Device listing

A client initiates a remote device listing by issuing an `OP_REQ_DEVLIST` request. This command consists only of a header that instructs the server to generate and return a device listing. Space for status value is always included in the protocol header but is set to zero in request headers. Status is only used to return status information from the server to the client. Figure 10 shows the `OP_REQ_DEVLIST` command with fields, the size of which represents the number of bytes each variable consumes.



1.) USB/IP user space protocol header

Figure 10. The `OP_REQ_DEVLIST` command, which is sent by clients requesting the list of available USB devices from the associated server.

A server replies to the previous device listing command by sending an `OP_REP_DEVLIST` reply. The reply is variable-sized and accommodates information about devices that are connected and bound to the server. The reply includes the protocol header and the number of devices, followed by device representations. A device is represented using a device information section followed by interface information sections. Device information contains general information about a device. This includes how it is connected to the server and contains basic information values as read from USB descriptors. The contained values include, for example, *idVendor* and *idProduct* numbers, which in theory define a specific device from a given vendor. This information is used to display to the user what devices are present and what capabilities a specific device has. The contents and structure of `OP_REP_DEVLIST` are presented in Figure 11.

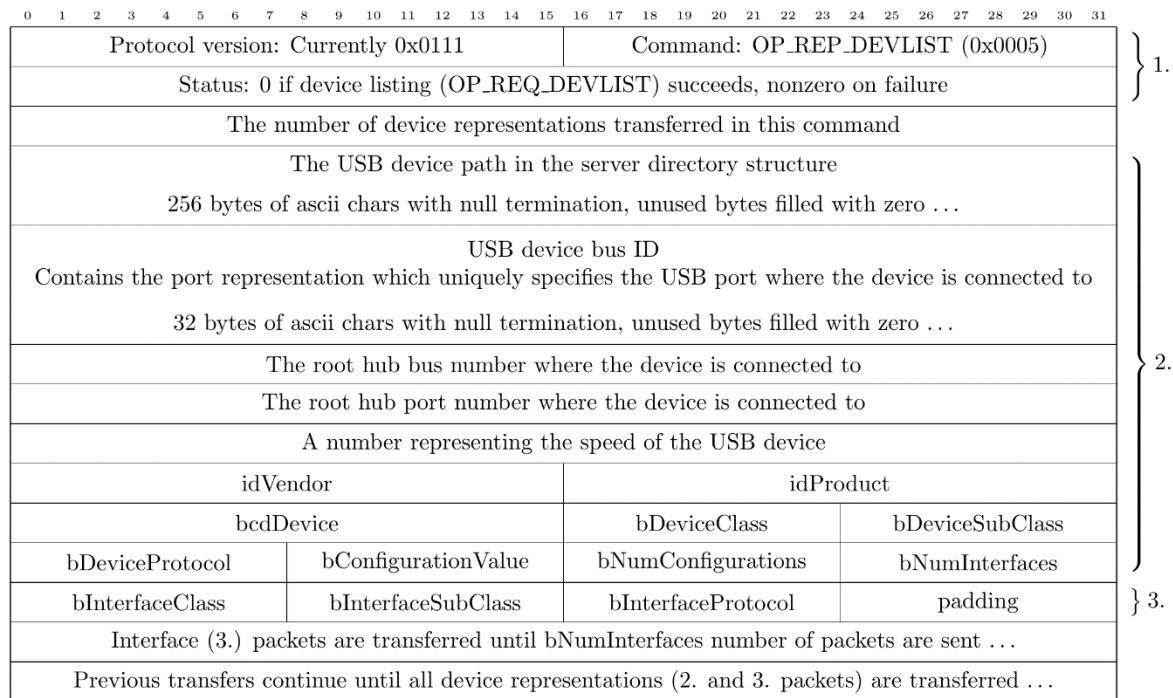


Figure 11. The OP_REP_DEVLIST command, which is the server's response to the OP_REQ_DEVLIST.

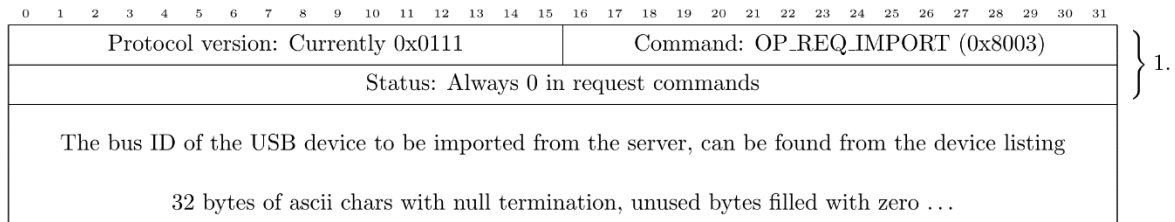
The device listing data returned during OP_REP_DEVLIST is formatted by the *usbip* executable so that it is suitable to be displayed for a user. Notably, although USB devices contain string descriptors for human-readable information, they are not used by the existing USB/IP system. Instead, the *usbip* tool displays the USB device names using a lookup file that matches *idVendor* and *idProduct* values with a list of known devices.

The *usbip* tool supports local device listings that are useful on servers to see information about locally present devices. These local listings are not transferred over the network and are useful on servers for finding devices to bind.

Attaching

A client starts the attaching process by selecting a device for use from the device listing information. From this data, the “bus id” value is selected and sent to the server, identifying the target device. The bus id is the unique combination of bus and port numbers that can be used to uniquely refer to a USB port on a server.

The request that allows a client to attach a USB device is OP_REQ_IMPORT, which includes the desired bus id. The server replies to the request with an OP_REP_IMPORT reply. The reply contains the success of the request and, if successful, also the device information that was present in the requested port at attach time. The following Figure 12 shows an attaching request that is used to indicate the desired bus id for the server. Figure 13 is the reply to the previous request.

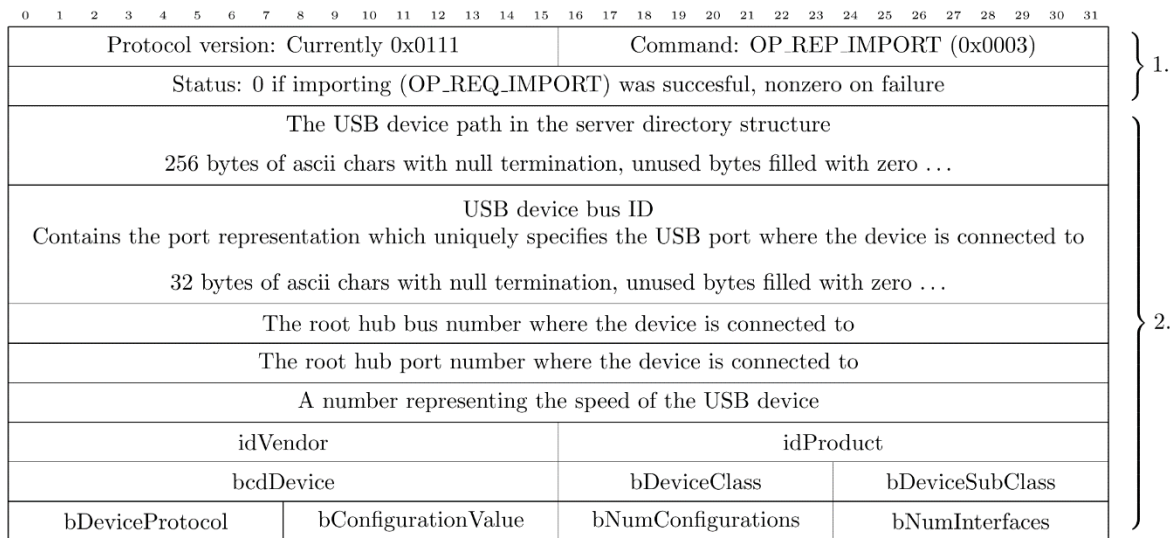


1.) USB/IP user space protocol header

Figure 12. A device is attached to a client with the OP_REQ_IMPORT command.

The reply to attaching request contains a header that tells a client whether the command was successful at the server end. If the execution was successful, the server has passed the open network connection to the STUB driver. After learning about this, the client will also pass its end of the socket to the USB/IP virtual host controller, and the protocol that is exchanged in the connection changes to the kernel space USB/IP protocol.

The control is passed to kernel drivers on the client by writing a sequence of the desired port number, open socket file descriptor, device id, and device speed to the *vhci-hcd* kernel driver provided *sysfs attach* file node. Likewise, the server uses the STUB driver provided *usbip_sockfd sysfs* file node to transfer the socket file descriptor. The passing event effectively equals plugging a new device into the client system.



1.) USB/IP user space protocol header

2.) Device information section

Figure 13. The OP_REP_IMPORT command, which is the server response to the OP_REQ_IMPORT request.

Detaching

Detaching a previously attached device requires no protocol communication over the network. When detaching a device, the *usbip* application on a client writes the virtual host controller port number where the target device is connected to the VHCI driver *detach sysfs* file node. This will then break the connection and remove the USB device from the client.

Binding

Servers bind and unbind devices with the STUB driver to enable and disable devices to be used with USB/IP tools. When a device is bound to the USB/IP driver, it is detached from the driver that was assigned to it during enumeration and, while bound, cannot be used by the server. When binding, the *usbip* application requests a bus id, which it then uses to unbind old drivers and assign the STUB driver. Binding requires no communication over the network.

4.2 Linux drivers

The USB/IP kernel drivers provide the means for USB remote use. On Linux, they expose their services through a filesystem interface for the user space USB/IP applications. This section presents the internal details, protocol, and functions of these drivers. While this chapter focuses on the Linux drivers, the protocol is the same on Windows as well.

The kernel module that is used by clients is the *vhci-hcd*, shortened to VHCI driver in this chapter, and the module used by servers is the *usbip-host* STUB device driver. The VHCI driver implements the virtual host controller and root hub functions. Servers utilize a STUB device driver that can communicate with any USB device. These drivers require a common *usbip-core* library, the use of which is transparent for users. The USB/IP kernel driver source code is in the Linux source code directory under *drivers/usb/usbip*.

All network TCP transfers between the kernel modules in USB/IP are done with Nagle's algorithm disabled. Nagle's algorithm is designed to buffer data and send it in larger chunks, which reduces network traffic since protocol headers do not need to be transferred as often [36]. However, this is critical to being disabled with USB/IP since it renders most devices unusable due to the increased delay and buffering.

Data in the USB cable has tight timing requirements, but it does not prohibit the use of USB devices with USB/IP since it does not deal with the USB physical layer. However, USB/IP and the network add delay to the USB transfers, and it can be a problem depending on the forwarded USB transfer type. For instance, some devices, such as high-definition USB web cameras, may be unable to be used at their maximum resolution. The effect of delay in the connection is investigated in the measurement chapter.

4.2.1 VHCI driver

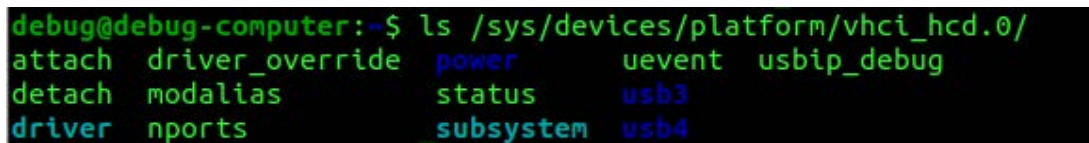
As explored in the USB chapter, USB transfers start from device drivers and travel through the USB core to a host controller and then onto USB peripherals. Host controllers implement a root hub which is used as the starting point for data transfers. With USB/IP, a software-defined virtual host controller is used instead of a physical host controller. This virtual host controller driver implements the root hub in software. Rather than generating USB cable traffic, the virtual root hub forwards URBs it receives to the USB/IP server.

The USB/IP VHCI driver implements the Linux USB host controller interface, which makes the virtual host controller appear like any other physical host controller as far as Linux is concerned. New devices are introduced to the virtual root hub using user space tools by writing to the driver *sysfs* attach node. Like with conventional host controllers and their root hubs, the Linux USB subsystem on the client computer polls the ports of the virtual root hub to learn information about new devices. The virtual host controller is transparent in

enumeration and data transfer and only routes all data to the server. The server STUB driver then completes the data requests through a regular USB stack.

Linux distributions such as Ubuntu, in general, include one VHCI driver instance that is capable of handling eight USB1.x or USB 2.0 devices and eight USB 3.0 devices. There is a compile-time configuration option to increase the allowed number of USB devices (USBIP_VHCI_HC_PORTS) to 30 in total (15 USB 2.0 capable + 15 USB 3 capable). To further increase the limit, there is a USBIP_VHCI_NR_HCS compile time option that allows up to 128 virtual host controllers.

The VHCI host controller kernel module implements services that user space host applications use. The VHCI driver's main *sysfs* functions are to allow checking the status of the root hub, attaching new devices, and detaching attached devices. These file nodes can be found in the folder `/sys/devices/platform/vhci-hcd.n` folder where the “n” indicates the index of the controller. While there can be more than one VHCI controller, common Linux distributions are compiled with only one enabled. Figure 14 presents the contents of the VHCI driver platform device node. The files *attach*, *detach*, *nports*, *status*, and *usbip_debug* are generated by the driver, and the rest are automatically generated by the Linux system.



```
debug@debug-computer:~$ ls /sys/devices/platform/vhci-hcd.0/
attach  driver_override  power          uevent  usbip_debug
detach  modalias         status         usb3
driver  nports           subsystem      usb4
```

Figure 14. Linux virtual host controller driver *sysfs* directory listing.

The *attach* node is a write-only file that is used to establish a virtual USB connection. It accepts a port number, a socket file descriptor with a connection to the server, a device id, and speed written into it. The driver validates the arguments, reserves a port from the root hub, and eventually starts USB device data transfers.

The *detach* node is also a write-only node, and it accepts a virtual root hub port number to be written into it. This will cause data transfers to be stopped, the socket to be closed, and the root hub port to become vacant. If the server computer devices are physically removed, they will also be automatically detached from the client.

The *nports* is a read-only node that returns the number of total root hub ports. The *usbip_debug* node allows reading or writing of a mask value that can be used to enable driver debugging features.

The *status* node is important because it is used to convey the internal status of the virtual host controller to user space applications. The *status* node is read-only and contains entries for each port of the VHCI root hub. It displays information about attached devices and the state of ports. Figure 15 presents an example of the contents of the *status* node.

```

debug@debug-computer:~$ cat /sys/devices/platform/vhci_hcd.0/status
hub port sta spd dev sockfd local_busid
hs 0000 006 001 00020005 000003 3-1
hs 0001 004 000 00000000 000000 0-0
hs 0002 004 000 00000000 000000 0-0
hs 0003 004 000 00000000 000000 0-0
hs 0004 004 000 00000000 000000 0-0
hs 0005 004 000 00000000 000000 0-0
hs 0006 004 000 00000000 000000 0-0
hs 0007 004 000 00000000 000000 0-0
ss 0008 004 000 00000000 000000 0-0
ss 0009 004 000 00000000 000000 0-0
ss 0010 004 000 00000000 000000 0-0
ss 0011 004 000 00000000 000000 0-0
ss 0012 004 000 00000000 000000 0-0
ss 0013 004 000 00000000 000000 0-0
ss 0014 004 000 00000000 000000 0-0
ss 0015 004 000 00000000 000000 0-0

```

Figure 15. Contents of the Linux virtual host controller status node showing one attached device to a high-speed root hub port.

The values returned by the status node are as follows:

- hub: The hub that is associated with the port (hs – USB 2.0 / ss – USB 3.0).
- port: The root hub's port number index.
- sta: Status of the port.
 - 4 – VDEV_ST_NULL: The port has no connection and is ready to use.
 - 5 – VDEV_ST_NOTASSIGNED: A connection is being established.
 - 6 – VDEV_ST_USED: The port is currently in use.
 - 7 – VDEV_ST_ERROR: The port is in an error state.
- spd: Speed of the connected device.
 - 0 – Unknown
 - 1 – Low-speed (1.5 Mbps)
 - 2 – Full-speed (12 Mbps)
 - 3 – High-speed (480 Mbps)
 - 4 – Wireless USB (WUSB)
 - 6 – SuperSpeed (5 Gbps)
- dev: Device identification number. The USB/IP user space tools derive it from the bus and address values which are assigned by the server.
- sockfd: The socket file descriptor number of an established connection.
- local_busid: These are the bus and port numbers on the client that were assigned to the virtual device.

4.2.2 STUB driver

The STUB driver implements the server part of USB/IP. It is a USB device driver running on a server and assumes full control of USB devices and their interfaces. It replaces previous device drivers that were automatically assigned to a USB device during enumeration. The

function of the STUB driver is to receive URBs as given by client-side device drivers and complete them through a conventional USB stack. The STUB is implemented in the *usbip-host* module, and the source files can be found in the Linux kernel source tree source directory *drivers/usb/usbip*. This section introduces the STUB driver and its functions.

STUB device driver *sysfs* entries become available for use after binding the driver with a USB device. The *usbipd* server daemon uses the STUB driver *sysfs* interface in its operation and passes the client-facing socket to the driver *usbip_sockfd* node to start USB data transfer. After the socket descriptor is given to the driver, the client and server can start exchanging the USB/IP tool protocol data. STUB also provides a *usbip_debug* node that can enable debugging features and a *usbip_status* node that reflects the internal status of the USB/IP state. The status can have one of the following values: 1 – SDEV_ST_AVAILABLE when the device is bound but not used by clients; 2 – SDEV_ST_USED when it is in use; or 3 – SDEV_ST_ERROR when a fatal error has occurred.

4.2.3 Protocol

The kernel space USB/IP protocol is used when exchanging USB data between VHCI and STUB drivers. In brief, the protocol consists of two client-initiated commands: USBIP_CMD_SUBMIT and USBIP_CMD_UNLINK. Both originate from the VHCI driver and allow submitting URBs and canceling previously sent uncompleted URBs. The STUB driver on the server replies to these commands with USBIP_RET_SUBMIT and USBIP_RET_UNLINK commands.

All commands are sent over the network in network byte order. This makes the USB/IP system processor architecture agnostic. The first 20 bytes of every protocol packet follow a standard structure defining a base header. The header is always followed by at least 28 bytes, which define a full 48-byte USB/IP command. There may also be a variable amount of data after a command that is used when reading or writing USB device data.

The base header consists of five 32-bit fields, which are:

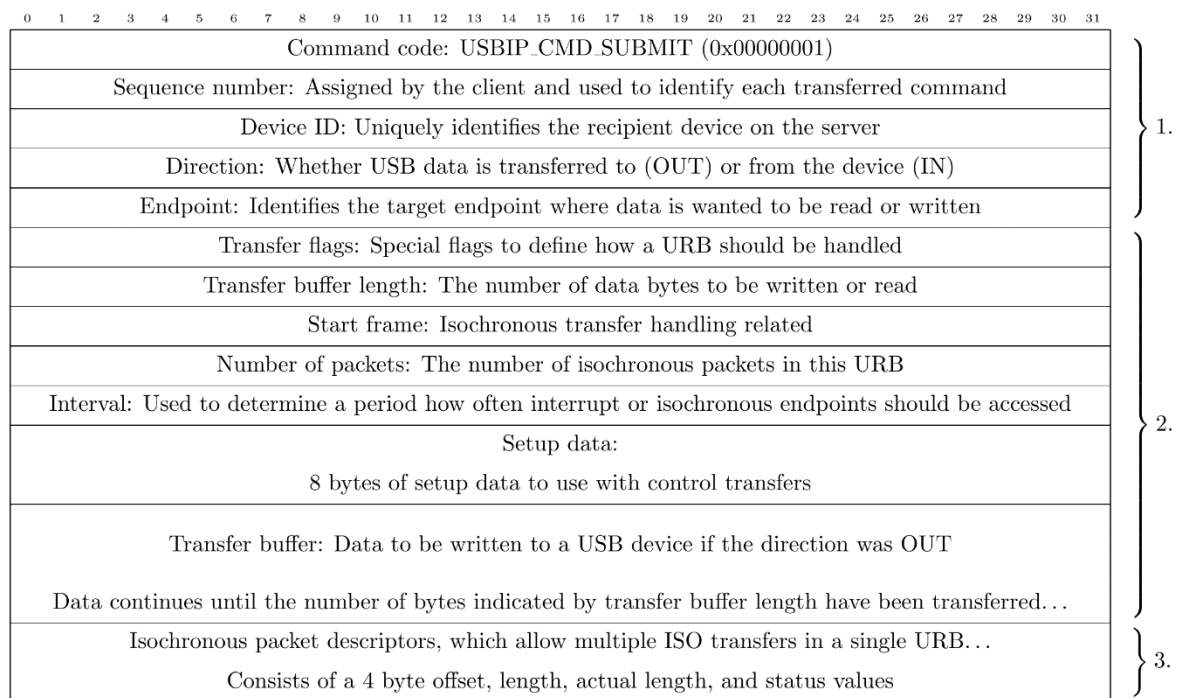
- Command type: Identifies the command following the base header.
- Sequence number: A rolling number that identifies each sent command. It is used when referring to packets at unlink time.
- Device id: A value that identifies the device that is referenced by the command.
- Transfer direction: Used to identify whether data is transferred to or from a peripheral.
- Target endpoint: Identifies the endpoint buffer to which the data transfer is targeted. It contains only the endpoint number, which is the four lowest bits of *bEndpointAddress* without direction information.

Not all header fields are relevant for every protocol command. Only the command type, sequence number, and device id are mandatory fields for all headers. The following sections introduce each of the protocol commands.

URB submission

The URBs originate from the client device drivers. The command that allows an URB to be sent from a client to the server is `USBIP_CMD_SUBMIT`, and the server responds after handling it with `USBIP_RET_SUBMIT`. This URB submission command is presented in this section.

The command `USBIP_CMD_SUBMIT` complements the previously shown base header with additional fields which are: transfer flags, transfer buffer length, start frame, number of packets, interval, and 8 bytes of setup data. These values are read from the original URB on the client-side. Figure 16 shows the contents of the full `USB_CMD_SUBMIT` command.



- 1.) USB/IP kernel space protocol base header
- 2.) `USBIP_CMD_SUBMIT` command payload and data
- 3.) ISO descriptors if transfer type was isochronous

Figure 16. The `USBIP_CMD_SUBMIT` command, which transfers URBs from the client VHCI driver to the associated server STUB driver.

The USB/IP tools support all four USB transfer types: `CONTROL`, `INTERRUPT`, `BULK`, and `ISOCHRONOUS`. The URBs that carry these USB transactions are used to read or write USB data as indicated by the transfer direction and endpoint they are sent to. The direction in the base header describes from the client's point of view whether data is sent from peripheral (IN) or to peripheral (OUT). The transfer type changes the internal handling of the command. The data flow of `CONTROL`, `INTERRUPT`, and `BULK` transfers is handled similarly, but the transfer direction causes a slight variation in the command handling. The network transfers and data flow of a `USBIP_CMD_SUBMIT` command with `CONTROL`, `INTERRUPT`, and `BULK` transfers are shown in Figure 17 from the server's point of view.

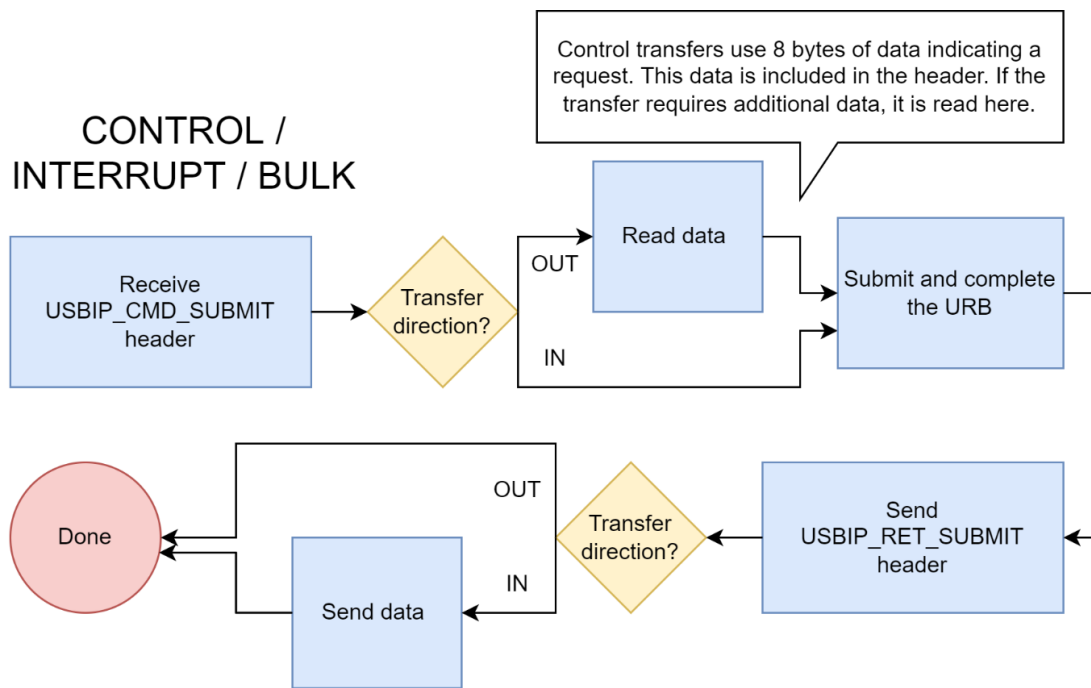


Figure 17. The USB/IP server handling of CONTROL, INTERRUPT, or BULK transfers.

Control transfers are sent to the default pipe, which handles both IN and OUT functions. However, the direction is not present in the endpoint value, and instead, the direction field is used to indicate this. Control transfers always use 8 bytes to represent a request, and these are included in the command header.

The client sets the direction IN to read data from a USB device. Otherwise, the client sets direction to OUT and transfer buffer size accordingly to indicate it wants to send additional data to be written to the device during USBIP_CMD_SUBMIT handling.

USB/IP carries information about how the URB should be handled when it is submitted. This information is included in the transfer flags, but not all existing flags apply to USB/IP.

Isochronous transfers have additional data transfers to carry isochronous (ISO) descriptors. These descriptors may contain information about many transfers and how data is structured in the transfer buffer. The following Figure 18 shows the structure of a USBIP_CMD_SUBMIT command in the case of isochronous transfer.

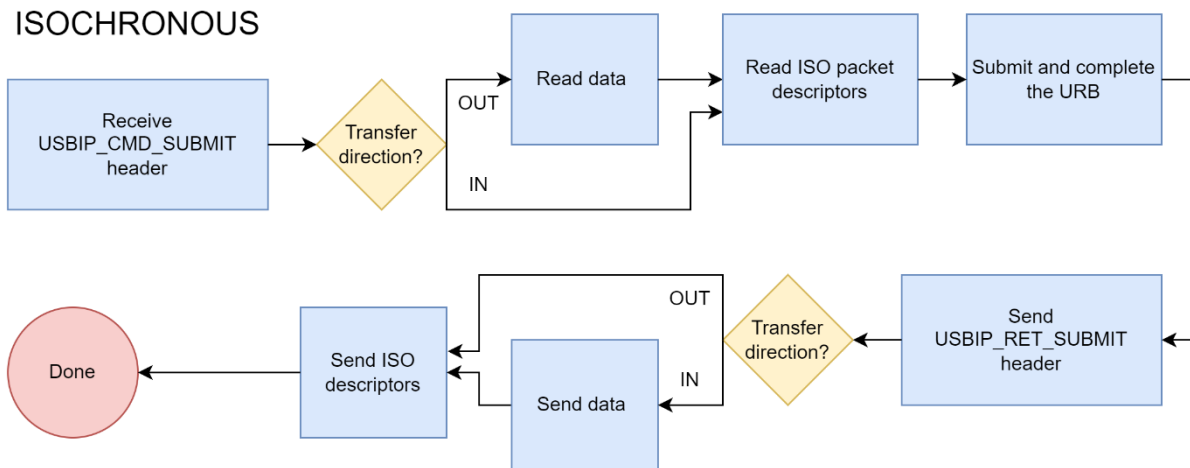


Figure 18. USB/IP server ISOCHRONOUS transfer handling.

A USBIP_RET_SUBMIT command is issued after a URB has been handled on a server. This command includes status, actual_length, start_frame, number_of_packets, and error count fields which reflect the values from completed URBs. If data was requested from a USB device, it is sent during the command handling. The status value employs standard Linux error codes and can contain information such as indicating device detachment or error conditions in endpoints. Figure 19 shows the contents of the USB_RET_SUBMIT command.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
Command: USBIP_RET_SUBMIT (0x00000003)																															
Sequence number: Of the corresponding USBIP_REQ_SUBMIT command																															
Should be set to 0																															
Should be set to 0																															
Should be set to 0																															
Status: Status code of the URB handling																															
Actual length: The amount of data read, can be different from what was requested																															
Start frame: Isochronous transfer handling related																															
Number of packets: Number of isochronous packets in this URB																															
Error count: Number of failed isochronous transfers																															
Should be set to 0																															
Should be set to 0																															
Transfer buffer: If URB direction was IN contains "actual length" bytes of data read from device																															
Transfer continues until actual length number of bytes are sent...																															
Isochronous packet descriptors, which allow multiple ISO transfers in a single URB...																															
Consists of 4 byte offset, length, actual length, and status values																															

- 1.) USB/IP kernel space protocol base header
- 2.) USBIP_RET_SUBMIT command payload and data
- 3.) ISO descriptors if transfer type was isochronous

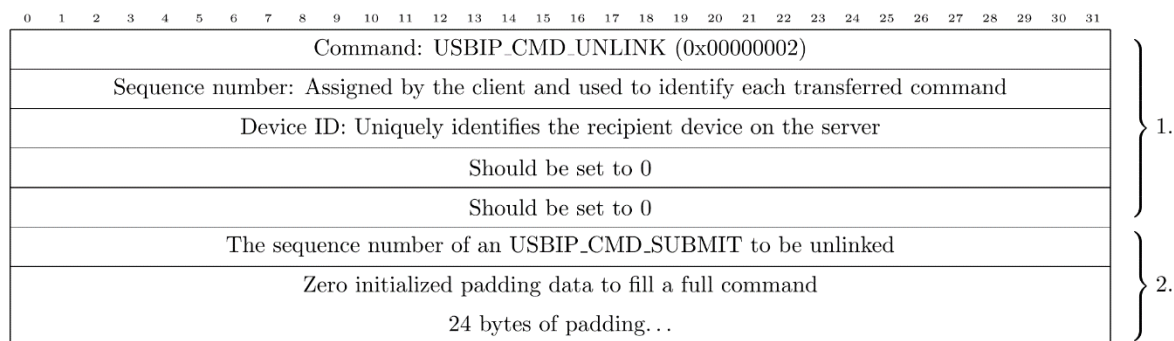
Figure 19. The USBIP_RET_SUBMIT command is a server response to the previously sent URB inside a USBIP_CMD_SUBMIT command.

As an exception, USBIP_RET_SUBMIT will not be sent in the case that a related USBIP_CMD_SUBMIT command was unlinked before completion. The unlinking causes related USB transfers to be canceled, and the command to support this canceling is presented next.

URB canceling

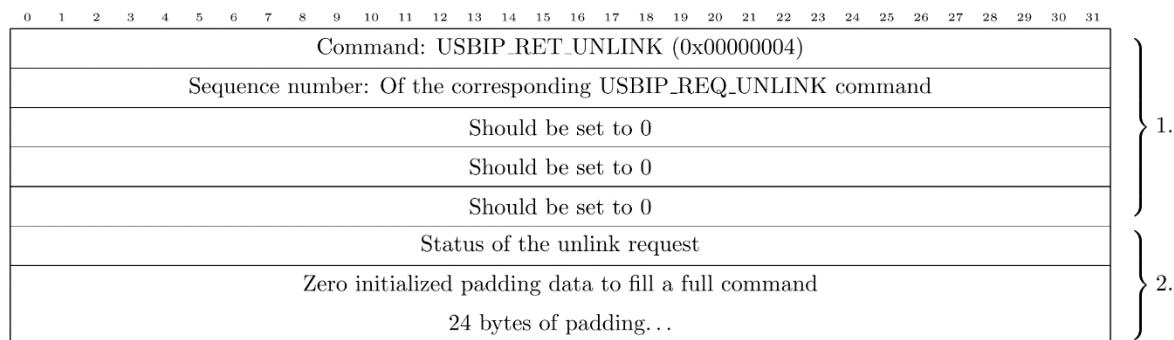
The USB system allows the cancellation of previously sent but not yet completed URBs. This feature is part of the required functions of a USB system as defined in the USB 2.0 specification. To support this feature, the VHCI driver sends a USBIP_CMD_UNLINK command that can be triggered by a call to *usb_kill_urb()* on a client device driver.

When the server receives this USBIP_CMD_UNLINK command, it attempts to find a previously sent USB/IP package with the given sequence id. If such a package is found, associated USB requests are canceled on the server and a USBIP_RET_UNLINK package is sent back to the client. When an existing USB/IP submit package was found and canceled, the related USBIP_RET_SUBMIT package is not sent to the client. Figure 20 shows the USBIP_CMD_UNLINK packet and Figure 21 shows the USBIP_RET_UNLINK packet.



- 1.) USB/IP kernel space protocol base header
- 2.) USBIP_CMD_UNLINK command payload

Figure 20. The USBIP_CMD_UNLINK command attempts to cancel a previously sent USBIP_CMD_SUBMIT and its associated URB.



- 1.) USB/IP kernel space protocol base header
- 2.) USBIP_RET_UNLINK command payload

Figure 21. The USBIP_RET_UNLINK command is the status of a previously sent unlink.

The status of an unlink also uses Linux error codes. As opposed to a conventional value of zero for a successful action, the unlink is successful when the status is *-ECONNRESET*.

5 IMPLEMENTATION

The Virtual I/O Box (VIOBox) project was used to evaluate whether a commercial USB sharing system could be built with open-source software and commercial off-the-shelf hardware. During this project, a prototype VIOBox device was built for evaluation and demonstration purposes. The hardware and mechanical aspects of this VIOBox prototype are first presented in this chapter for general background. The creation of a new RemoteHub USB device remote sharing tool for use in VIOBox was the goal of this project and is presented after the general device overview.

5.1 VIOBox device

The VIOBox device is comprised of a commercial-off-the-shelf Single-Board Computer (SBC) integrated into an Aava tablet charging station. This section presents the hardware choices in VIOBox.

5.1.1 Single-board computer

Two alternative SBCs were evaluated to be used with VIOBox. These were a Raspberry Pi 3 Model B and an Orange Pi Zero. These are referred to as the Raspberry Pi and the Orange Pi in the following text. The main capabilities of each are listed below in Table 3. Both SBCs were in the desired price range and were expected to suit the needs of VIOBox. Overall, comparing the features of the evaluated SBCs, the Raspberry Pi was more capable. However, it also included features such as HDMI and camera connectors, which were not needed in the VIOBox. [37][38]

Table 3. High-level comparison of evaluated SBC devices

	Orange Pi Zero	Raspberry Pi 3 Model B
Price at the start of the project	~15€	~30€
Processor	Allwinner H2+ (ARM)	Broadcom BCM2835 (ARM)
Memory	512 MB	1 GB
Storage medium	microSD card	microSD card
USB port count	3 (With extension shield)	4
Integrated ethernet speed	100 Mbps	100 Mbps

There are three USB ports in an Aava Mobile docking station. Therefore, VIOBox was also required to have at least three integrated USB ports. The Raspberry Pi had four USB ports integrated, but the Orange Pi contained only one. One solution to add more ports would have been to use a standard hub connected to the SBC. The Orange Pi also allowed extending the ports with an extension card module. The Orange Pi extension module used USB traces that were routed to pins on the SBC board. However, this card also included redundant features for VIOBox, and the USB ports were in the opposite direction of the existing port. For this reason, a custom USB port extension solution was built for the Orange Pi.

Both SBCs contain an Advanced RISC Machines (ARM)-based Central Processing Unit (CPU). The Orange Pi uses the Allwinner H2+, which contains four Cortex-A7 cores. The Raspberry Pi includes the Broadcom BCM2837, which has four Cortex-A53 cores. The 32-bit

Cortex-A7 was introduced in 2011 and supports the ARMv7 instruction set. The Cortex-A53 supports 64-bit ARMv8 instructions and is the successor to the A7. Considering raw performance, the processor in the Raspberry Pi is more performant.

A hardware-based cryptographic accelerator could help boost TLS performance. However, neither of the two computers had true ARM cryptographic extensions. However, the Orange Pi includes a crypto engine that, among other ciphers, supports the AES algorithm. This extension is implemented as a proprietary intellectual property block that requires software support and may only be of limited help [39].

Both SBCs support Linux-based operating systems. More specifically, in this project, 32-bit Raspbian was used on the Raspberry Pi and 32-bit Armbian on the Orange Pi. While the processor in the Raspberry Pi supports 64-bit instructions, the support in Raspbian has been implemented only recently. The Raspberry Pi foundation has used 32-bit images so that they can be used on all Raspberry Pi devices, avoiding customer confusion. Performance-wise, there is some improvement using the 64-bit instruction set, but this is currently mostly visible during benchmarking and not in real-world use [40].

Both computers were tested as the computer in VIOBox. Orange Pi was decided to be the better option. It was more cost-efficient and fit into the existing data module dimensions.

5.1.2 *Mechanical construction*

The VIOBox was designed to appear similar in comparison to other Aava Mobile docks. For reference, Aava has two types of tablet docks: charging-only docks and combined data and charging docks. The charging-only dock contains charging pins that connect to Aava tablets and has the same external dimensions as the data dock. The combined data dock includes an additional module that adds a traditional three-port USB hub and an HDMI port.

The first prototype VIOBox used the Raspberry Pi, and a case for it was 3D printed. This case was connected externally to the dock because of the dimensions of the Raspberry Pi. Figure 22 depicts this device. The second prototype was based on the Orange Pi and was much more compact. The second prototype module filled the extension module opening at the back of the dock, visible in Figure 22.



Figure 22. The VIOBox dock with the Raspberry Pi 3 Model B connected as an external module.

5.2 Software requirements and design choices

When exploring existing open-source software tools, it was found that they would not be suitable for VIOBox as is. Existing USB/IP tools were limited in the necessary data security and automatic use aspects. Therefore, new software was decided to be built, and this section describes the initial software planning. First, VIOBox software was assigned core requirements that reflect the functions that needed to be supported. These were used as a guideline for development. Requirements were split into two categories, functional and non-functional requirements. Functional requirements define what the system should do, and non-functional requirements define how the system should operate [41]. The requirements were given priorities in the spirit of RFC 2119 [42], which reflects their importance in VIOBox.

The following is a list of functional requirements:

- The system **MUST** support hot plugging, which means USB devices are automatically exported from the server to the client. The client starts automatically using USB devices when they are attached to the server.
- The client **MUST** be able to find servers automatically on the local network. When a client detects that a suitable server is present in the network, it can automatically start communicating with it.
- When there are multiple servers in the same network, the client **MUST** have the capability to distinguish them and have the capability to determine which servers to use.
- The server **MUST** hide certain USB devices so that they are not visible to clients. These include, for example, internal USB network adapters or other private devices.
- The client **MAY** be able to use the server manually. A connection can also be established with a server in cases where automatic discovery cannot be used, such as over the public internet.

The following is a list of non-functional requirements:

- Transferred data between server and client **MUST** be encrypted. This includes both the protocol commands and USB data.
- The server application **MUST** support the Linux environment.
- Applications **MUST NOT** require manual configuration after initial setup. After the applications have started, they work fully autonomously.
- The applications **MUST** support initialization with configuration files.
- The software **MUST** support at least three USB devices simultaneously.

- The USB devices **MUST** be attached in a reasonable time (<10s) after plugging them into the server computer.
- The applications **SHOULD** provide a command-line-based user interface for debugging. The server and client print information about their internal operation.

The previously introduced requirements could be addressed in VIOBox software by creating scripts for existing USB/IP tools. The scripts would need to control all the necessary separate applications for implementing the requirements. However, this was determined not to be optimal due to maintainability and the general feel of the system. Another option would have been to modify the existing USB/IP tools and implement the required functions using software libraries. However, VIOBox would benefit from a system that is built to prioritize the new requirements. Building a new set of tools would allow good flexibility with the software execution flow and future improvements. Therefore, the server and client were rebuilt with the VIOBox use case in mind in this project.

The new set of tools, named RemoteHub, implement the previously introduced requirements and are used in the prototype VIOBox device and Aava Linux tablets. The client application uses the existing USB/IP virtual host controller kernel driver like the *usbip* tool does, but the *usbipd* and *usbip-host* functions are both implemented in user space. Although user space STUB server implementation was cautioned by Hirofuchi to introduce memory copy overhead and is the main reason current USB/IP drivers exist in kernel space [6], it would allow for better maintainability and portability for future expansion. If needed in the future, RemoteHub can be extended with capabilities such as a graphical user interface or fine-grained user access controls.

From a technical standpoint, RemoteHub implements data encryption and verification actions with TLS and uses a custom UDP broadcast solution for server discovery. The existing USB/IP protocol is utilized to make use of the USB/IP VHCI driver and ease possible future development with Windows USB/IP tools. The following sections describe the implementation choices in more detail.

5.2.1 Security

RemoteHub uses the TLS protocol with certificates to provide data security between the client and server applications. The fundamental background of TLS was presented in the second chapter. The library in use is MbedTLS by TrustedFirmware [43], which is a well-documented library suitable for C applications. MbedTLS is available on both Linux and Windows. This library was also tested with a proof-of-concept Windows USBIP-win implementation. MbedTLS claims to have been purpose-built for resource-constrained devices, which should be helpful considering the hardware that VIOBox uses as a prototype and in the future. Testing was conducted to find a cipher suite with good performance, and the results are presented in the sixth chapter. MbedTLS supported up to TLS 1.2 at the time of implementation, and a subset of the TLS version 1.2 compatible cipher suites were used in the evaluation. RemoteHub was built to optionally support unencrypted TCP transfers without data security features to allow performance comparisons. This was done by adding an abstraction layer that allows mostly unified data transfer without knowledge of the underlying transport method. Due to this implementation, assessing different TLS libraries in the future is expected to be straight-forward.

5.2.2 *Automatic discovery and unattended use*

The RemoteHub server advertises its existence by periodically sending UDP broadcast packets. These packets are not exchanged reliably but are configured to be sent once every five seconds. The lack of reliability is acceptable since a client is still expected to receive most of these packets. The UDP approach requires no knowledge from the server of clients in the network, which facilitates the automatic operation. When a server sends this package to the router, it is broadcast by the router to all other devices on the same network. Clients can either ignore these packets or establish a connection automatically with the help of the accompanying server information. The information in these packets includes the server's name, TLS support, and the port that is listening on the server. Clients also extract the server's IP address from the packet.

The automatic discovery enables VIOBox to be used unattended. Because VIOBox contains no real user interface, this was a necessity. Another implemented feature was hot plugging. This means all devices that are plugged into VIOBox are automatically taken into use by the client. This was desired so that the system resembles a conventional USB hub.

5.3 RemoteHub library and application design

As described previously, RemoteHub includes the VIOBox-specific technical requirements regarding data security and automatic use. The RemoteHub server application is a Linux executable that bundles the functions of the USB/IP STUB driver and *usbipd* connection daemon into one package. The server application implements the STUB as a user space driver with libUSB, which uses the Linux USB device driver interface internally. User space implementation allows better maintainability and debugging options [28], and supports easier portability to other platforms if needed in the future. The existing USB/IP and the new RemoteHub are compared in the sixth chapter.

USB devices are not bound to the server STUB driver in RemoteHub but are rather automatically taken into use by clients at run time. To support automatic operation, device listing and attaching requests are automatically executed using the previously explored USB/IP user space protocol. Internally, RemoteHub tools are written in the C language. Figure 23 shows the internal data flows and how they fit into the Linux USB system. RemoteHub is an open-source project freely available on GitHub [44]. It can operate on a wide range of devices despite being designed for VIOBox.

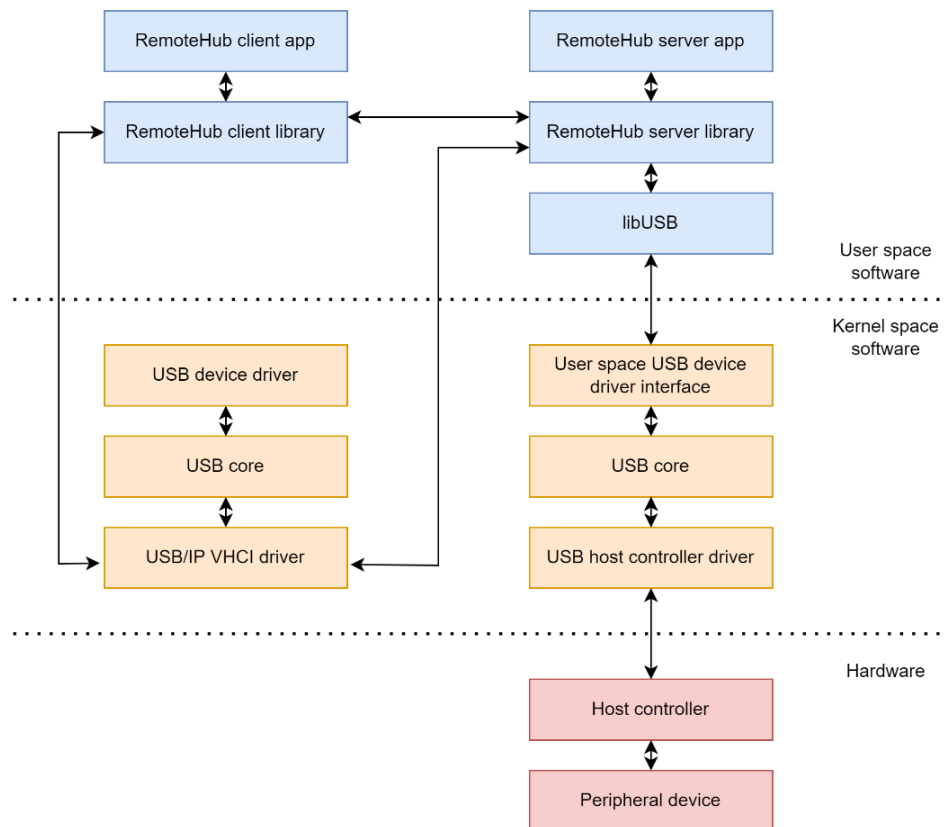


Figure 23. Illustration of the RemoteHub hierarchy of data flows on two separate computers (client on the left and server on the right).

This section presents the internal design choices of RemoteHub applications and libraries in more detail. Most of the application logic is available through the RemoteHub libraries, which the applications use through an API. There are specially crafted libraries for both server and client use cases. They provide most network and USB functions in an asynchronous manner. This means that most network-related client commands do not block, and instead a callback function is invoked after the library has executed the command.

Internally, the libraries consist of multiple separate tasks that work independently. The tasks in the libraries communicate with each other using events. The events are delivered by type to tasks that are subscribed to receive them. This design is referred to as the Publish-Subscribe pattern [45]. This pattern allows for loose coupling between tasks, which encourages code re-use and can make the codebase more maintainable. The following Figure 24 shows the internal tasks and data flows in both client and server applications.

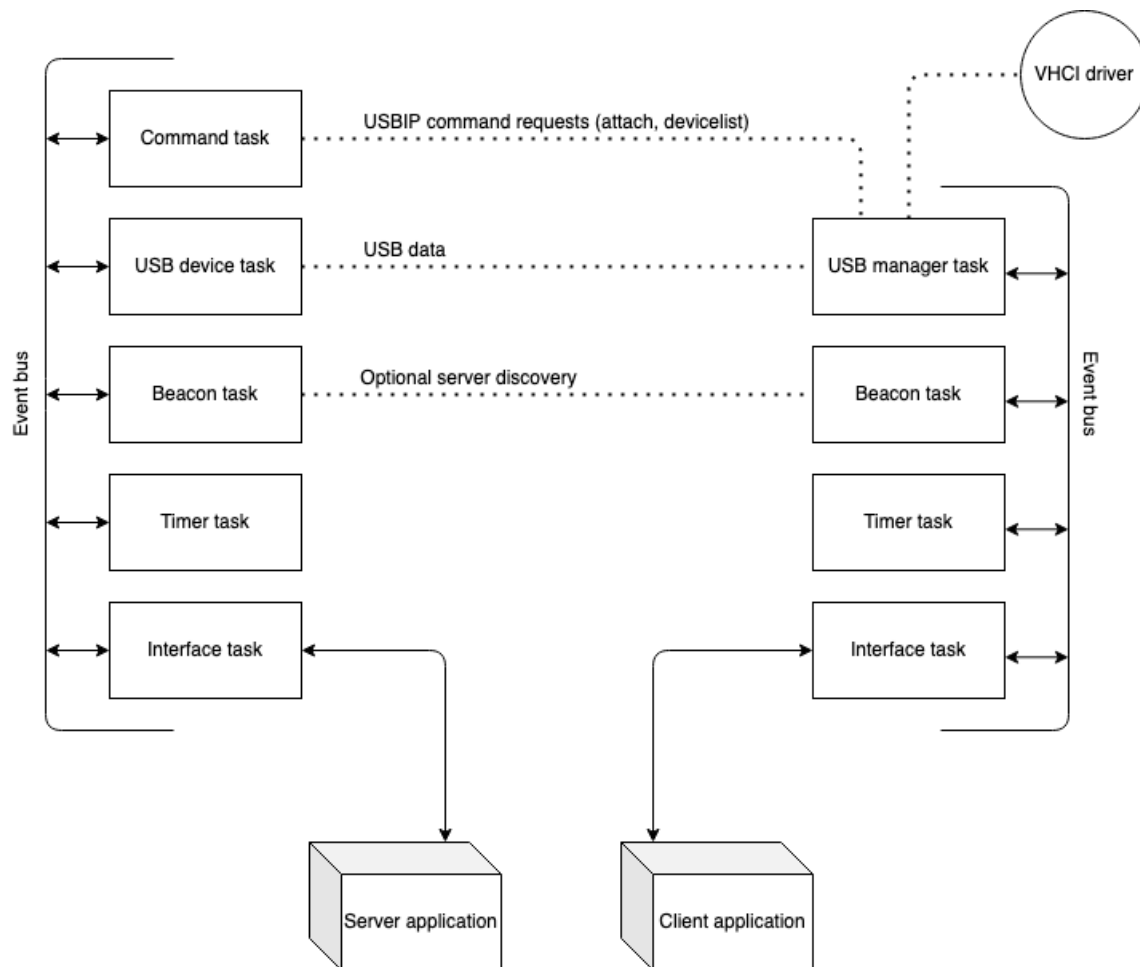


Figure 24. RemoteHub high-level architecture diagram.

All created source code was written to comply with the kernel coding style. The kernel coding style by Linus Torvalds provides a good set of rules for maintainable C code [46]. Coding style preferences vary widely among coders, but keeping a single coding style throughout the codebase allows readers to focus on the program logic rather than varying syntax.

5.3.1 Libraries

RemoteHub itself consists of three core libraries. There are separate libraries for server and client use cases, and a third library that contains common functions for both use cases. There are also third-party libraries that implement the USB transfer, TLS, and JavaScript Object Notation (JSON) features. Most libraries are designed to be statically linked, which means they are archives containing executable code. This way, the final compiled application executables contain most of the necessary functions built in. One exception is the libUSB library, which is dynamically linked. Dynamic linking reduces the executable size by allowing multiple programs to use the same library that is stored only in a single location. Users of the RemoteHub server application need to have this libUSB library available to be able to use the program.

Common library

The common library provides functions that are used by both client and server libraries internally. These functions include event driving, network, and debugging utilities. The event framework in the common library facilitates the transfer of information between tasks that the server and client libraries consist of. The event framework is identical for server and client libraries and is therefore included in the common library. Events are distinguished by the type identifier that is set when enqueueing them. They can also carry arbitrary data using a pointer. The framework and tasks use the *pthread* library extensively for threading and synchronization.

Tasks in the libraries are first registered with the event framework by calling the *event_task_register()* function. This function is called with a bit field of events the task is interested in. The task starts a new thread for receiving these events. While running, the task can produce events for other tasks. The framework will handle the event delivery logic internally, providing *event_enqueue()* and *event_dequeue()* functions for convenience. The *event_dequeue()* function blocks when no events are available for a particular task. When the system is wanted to be terminated, the stopping is facilitated with a terminating event. This will cause all tasks to finish their execution, allowing the system to be brought down in a controlled manner. The following Figure 25 presents a simplified lifecycle of a task, including event handling related functions.

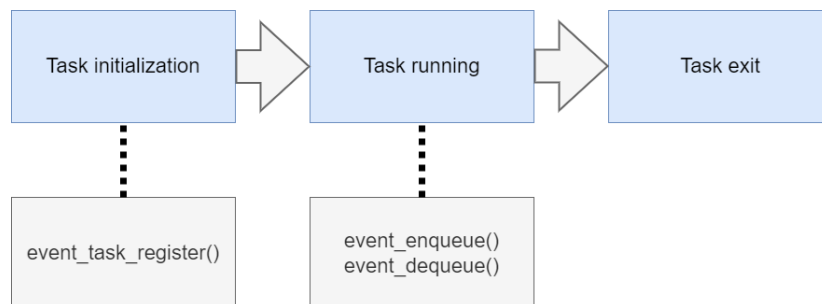


Figure 25. The life cycle of a task in RemoteHub.

The common library contains a synchronized debug printing facility for printing information about the internal operation if enabled. There are six debug levels available that print information with increasing verbosity. The levels are presented in Table 4. Debugging can be enabled by calling the function *rh_set_debug_level()* with the desired level. All traces above the target level are also printed when selecting a debug level. For example, the “Trace” level prints all possible debugging traces.

Table 4. Debugging log levels in RemoteHub

Debug level	Description
Critical	A fatal error has occurred, and the execution cannot continue. This terminates the execution forcefully with the SIGABRT signal after printing the error.
Error	Something has failed, but execution may be able to continue. For example, failing a connection with a client would be printed as an error.
Warning	Something has failed, but execution can continue. For example, if some operation is not supported in a specific system.
Information	Traces that are expected to be informative.
Debug	Traces that are expected to be less relevant or only relevant in a specific section of the code.
Trace	Anything else, can be very verbose. For example, printing the received USB/IP command data.

Client library

The client library is the backbone for client applications. The main functions of the library are to find servers, list server devices, and attach them. It uses the existing USB/IP VHCI driver internally for USB data transfer and host controller functions. The library acts as a proxy in the USB data transfer process. Proxy threads are created for transferring encrypted data over one link and unencrypted data with the VHCI driver over another. The socket that is used with the VHCI driver is a stream-oriented UNIX domain socket that is ideal for inter-process communication.

The client library is made of four tasks that communicate with each other to produce services for the client application. Table 5 shows these tasks and the events they produce and consume.

Table 5. Event flows in the client library

Task	Produces events	Consumes events	Description
Manager	DEVICELIST_READY DEVICELIST_FAILED DETACHED DETACH_FAILED ATTACHED ATTACH_FAILED	TIMER_5S DEVICELIST_REQUEST ATTACH_REQUESTED DETACH_REQUESTED	Sends USB/IP requests to the server and manages imported USB devices.
Beacon	SERVER_DISCOVERED	-	Receives server beacon packets.
Timer	TIMER_5S	-	Creates timing events for other tasks.
Interface	DEVICELIST_REQUEST ATTACH_REQUESTED DETACH_REQUESTED	SERVER_DISCOVERED DEVICELIST_READY DEVICELIST_FAILED DETACHED DETACH_FAILED ATTACHED ATTACH_FAILED	Provides asynchronous callbacks for applications.

The manager task is responsible for initiating connections with servers as requested by the client application. It uses the USB/IP user space protocol to fetch device lists and attach devices. Network connections that are initiated by the manager support both TLS-secured and unencrypted transfers by using an abstraction layer that provides a unified interface for both. However, only either one can be used after the library has been started. Figure 26 illustrates this new connection initialization process in the client.

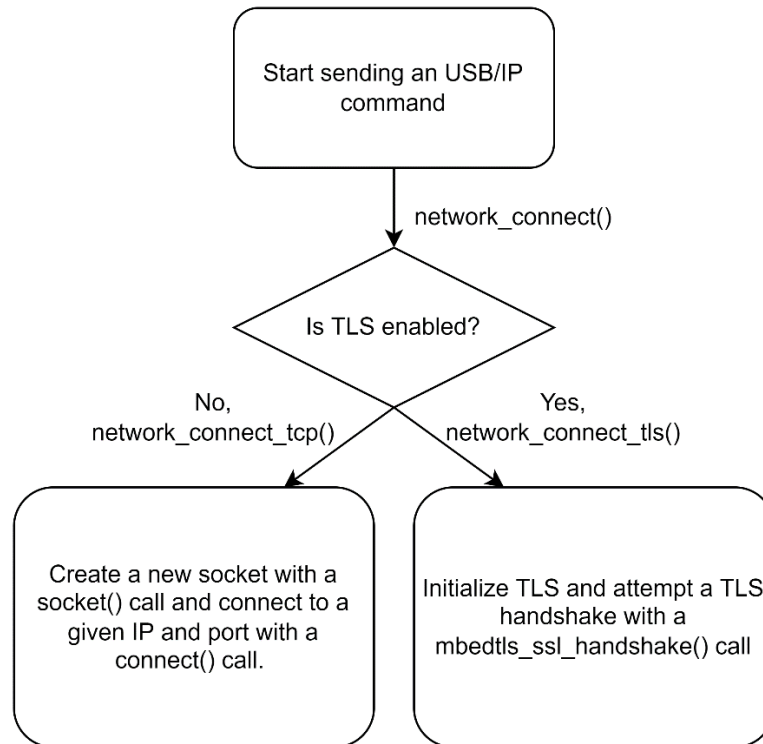


Figure 26. The RemoteHub client network connection process.

The manager is responsible for keeping track of attached devices in a linked list so that they can be safely found when needed. It also starts and oversees the data proxy threads for communicating with the VHCI driver. The beacon task supplies information about detected servers to the client application. After the client application receives this information from a callback function, it can attempt to connect to the server in question. The timer task is the simplest task available. It only generates 1 s and 5 s timing events, which are used by other tasks. The interface task acts as the asynchronous callback provider for applications. Like in the server library, it receives events from other tasks and passes them to the applications. The interface layer allows applications to toggle individual callbacks in an orderly fashion.

Server library

The RemoteHub server library implements the STUB driver in user space and allows a server application to monitor the server's internal state. The server implementation is a difference compared to existing USB/IP tools by Hirofuchi, which implement the STUB portion in kernel space. The implemented STUB server communicates with client VHCI drivers using the previously presented USB/IP kernel protocol. The USB transfers are routed through the

RemoteHub client library proxy to enable the encryption. However, the server is compatible with existing USB/IP tools without encryption.

Like the client library, the server library also consists of tasks. The server library implements five distinct tasks, which are presented in Table 6 with the flow of events.

Table 6. Event flows in the server library

Task	Produces events	Consumes events	Description
Command	REQ_DEVICELIST REQ_IMPORT	-	Receives USB/IP commands from clients.
Beacon	-	TIMER_5S	Broadcasts presence for clients in 5s intervals.
Timer	TIMER_5S TIMER_1S	-	Creates timer events for other tasks.
Interface	-	LOCAL_DEVICELIST DEVICE_ATTACHED DEVICE_DETACHED DEVICE_EXPORTED DEVICE_UNEXPORTED	Provides the asynchronous callbacks for applications.
USB device	LOCAL_DEVICELIST DEVICE_ATTACHED DEVICE_DETACHED DEVICE_EXPORTED DEVICE_UNEXPORTED	REQ_DEVICELIST REQ_IMPORT TIMER_1S	Handles USB device-related functions such as detection and data transfer.

The command task receives USB/IP commands sent by clients. The valid commands are for device listing and attaching using the USB/IP user space protocol. It can receive either TLS or unencrypted TCP communications as configured during initialization. As with the client, network access is abstracted to hide connection methods.

The beacon task generates UDP broadcast packets periodically for clients. These packets are sent to a special broadcast address, which causes them to be propagated to all the devices on the same network. In the initial version, the interval is 5s.

The timer task is essentially the same that is used on the client; it generates periodic timing events for beacon and other tasks. Also, like on the client, the interface task provides asynchronous callbacks for server application executables. Applications subscribe to being notified when an event of interest is received by using callback functions. The interface uses mutual exclusions, which makes all access thread-safe.

The USB device task is the most complex task. It drives the user space STUB driver and handles USB device housekeeping and discovery functions. It relies on the libUSB library for all USB-related functions and transfers. libUSB is a multi-platform library for accessing USB devices and allows asynchronous USB data transfers on Linux. In the initial RemoteHub version, USB devices are discovered using periodical polling. Upon detection of a new device, the task stores a reference to it in an internal linked list of present devices for future use. This is needed to send device listings to clients and facilitate a graceful exit of the application. The internal view of a USB device includes USB descriptor information, which is

read during the initial detection phase. As opposed to the existing USB/IP tools, which display device and manufacturer names using a lookup file, the RemoteHub server reads these from the USB device itself.

USB device insertion and removal actions generate events that can be captured by the server application and can be used for logging or other purposes. Internally, the included STUB driver starts transmission and receiving threads for each remotely used USB device. Using these threads, the STUB handles USB/IP protocol `CMD_SUBMIT` commands as shown in Appendices 1 and 2 and `CMD_UNLINK` commands as presented in Appendix 3.

5.3.2 Applications

Both server and client applications are built using an API that is defined by the RemoteHub libraries. It is important to note that the information here applies only to the initial version of RemoteHub. The RemoteHub libraries follow the semantic versioning scheme [47], and as long as the major version number is zero, anything in the API can change.

The applications and the libraries are built using the *CMake* [48] build automation toolkit. *CMake* can create build scripts for several platforms. This support is good to have for possible future development, although RemoteHub initially only supports Linux.

Client application

The RemoteHub client executable is built from the *rh_client.c* source file. The client communicates with the server to import USB devices that are present on the server. All network communication is abstracted in the libraries, which the applications control through the API. Like the server application, the client is a standard C program and supports Linux systems. Only the most relevant aspects of the application are covered in this section. A minimal usage example of the client application is shown in Figure 27.

```
int main(int argc, char *argv[])
{
    signal(SIGINT, sig_handler);
    . . .
    rh_client_config_init(conf_path);
    . . .
    while (running)
        usleep(5000000);

    rh_client_exit();
    return 0;
}
```

Figure 27. Minimal client *main()* function where error handling has been omitted for brevity.

The *main()* function is the starting point for execution. First, the *signal()* function is called during initialization, which registers a signal handler named *sig_handler()*. In the RemoteHub use case, only the SIGINT signal is caught. This signal is generated, for example, with a CTRL+C key combination. Catching this signal allows the program to shut down gracefully

because it enables the *rh_client_exit()* function to be called before exiting. This function only exits after all internal tasks have been shut down and internal memory reservations have been released. The example signal handler is presented in Figure 28.

```
void sig_handler(int sig)
{
    running = false;
}
```

Figure 28. An example of the signal handler function.

The library is started and terminated by calling the *rh_client_config_init()* and *rh_client_exit()* functions. The *rh_client_config_init()* function accepts a path to a JSON configuration file which, among other configuration options, contains the CA certificate path that will be used in the server verification process.

The application in

Figure 27 only starts the library, and additional logic is needed to start using the remote USB devices. The automatic operation is established by using callbacks that create a sort of pipeline that is automatically executed. Once a server is discovered, a device listing is immediately requested. If the command was executed successfully, all devices indicated by the listing are attempted to be attached. Finally, an attach callback is invoked when a device has been successfully connected.

Figure 29 shows the function calls that are used when subscribing to events, and Figure 30 shows the callback function skeletons that would be called when the respective events have fired.

```
rh_usbip_devicelist_subscribe(usbip_devlist_callback);
rh_attach_subscribe(attach_callback);
rh_detach_subscribe(detach_callback);
rh_server_discovery_subscribe(on_server_discovered);
```

Figure 29. The function calls that are used to subscribe the client application to library notifications.

```

void attach_callback(bool success, char *server, uint16_t port,
                    struct usbip_usb_device dev)
{
    . . .
}

void detach_callback(bool success, char *server, uint16_t port,
                    struct usbip_usb_device dev)
{
    . . .
}

void usbip_devlist_callback(bool success, char *server, uint16_t port,
                           struct usbip_usb_device *devlist, uint32_t count)
{
    for (uint32_t i = 0; i < count; i++)
        rh_attach_device(server, port, devlist[i]);

    rh_free_client_devlist(devlist);
}

void on_server_discovered(char *server_ip, uint16_t port, char *name)
{
    rh_get_devicelist(server_ip, port);
}

```

Figure 30. The function skeletons with a sample logic flow that will be invoked by the RemoteHub client library.

Initially, the devices that are not desired to be used are only detected and skipped in the device listing callback *usbip_devlist_callback()*. The device listing callback is called with a list of USB device representations on the server. This is the information that is transferred with the USB/IP user space protocol. Additionally, if automatic server discovery is not desired, the *rh_get_devicelist()* function can be manually used to connect to a server. However, the auto-discovery is always needed when using VIOBox.

Server application

This section presents the RemoteHub server application implementation. The executable is built from the *rh_server.c* source file. Only the most relevant aspects of the implementation are covered here. A simplified application logic flow is presented in Figure 31, which closely resembles the client application.

```

int main(int argc, char *argv[])
{
    signal(SIGINT, sig_handler);
    . . .
    rh_server_config_init(conf_path);
    . . .
    while (running)
        usleep(100000);

    rh_server_exit();
    return 0;
}

```

Figure 31. Minimal server *main()* function where error handling has been omitted for brevity.

The *rh_server_config_init()* function is used to start the server library. Like the client application, it accepts a path to a JSON-formatted configuration file. This file contains important instructions such as the port to use and the paths of the server certificate and private key for TLS connections. The contents of this file are presented later in this chapter.

During initialization, the server subscribes to necessary asynchronous notifications generated by the library. The application will be notified each time a USB device is physically attached or detached if *rh_attached_subscribe()* and *rh_detached_subscribe()* have been used to subscribe a callback to the events. The *rh_exported_subscribe()* and *rh_unexported_subscribe()* registered callbacks are called when a client attaches or detaches a device from the server. All these functions accept a function with the same signature. The function *rh_devicelist_subscribe()* is used to subscribe to local device listings that are only needed for the user interface. Figure 32 shows these functions being used to assign *device_state_changed()* and *devlist_handler()* functions for the events. Figure 33 shows minimal implementations of these functions.

```

rh_devicelist_subscribe(devlist_handler);
rh_attached_subscribe(device_state_changed);
rh_detached_subscribe(device_state_changed);
rh_exported_subscribe(device_state_changed);
rh_unexported_subscribe(device_state_changed);

```

Figure 32. The calls that are used to subscribe the application for library notifications.

```

void device_state_changed(enum usb_dev_state state,
                        struct usbip_usb_device dev)
{
    . . .
}

void devlist_handler(struct usb_device_info *devlist, int count)
{
    . . .
    rh_free_server_devlist(devlist);
}

```

Figure 33. The function skeletons which are invoked by the RemoteHub server library after previous subscription.

In the previous functions, the *enum usb_dev_state* can be one of: ATTACHED, DETACHED, EXPORTED, or UNEXPORTED and reflects the event that generated it. The structure *usbip_usb_device* contains USB device information that is transferred with the USB/IP user space tool protocol. The structure *usb_device_info* contains additional data such as the manufacturer name, but also includes the *usbip_usb_device* information internally.

5.3.3 Licenses

This section presents the licensing information of the third-party libraries in RemoteHub and how these can affect the created software. License types of the used libraries are presented in Table 7.

Table 7. Licenses of software libraries used in RemoteHub

Library	License
libUSB	GNU's Not UNIX (GNU) Lesser General Public License version 2.1 (LGPLv2.1)
MbedTLS	Apache License, Version 2.0
cJSON	Massachusetts Institute of Technology (MIT) License

GPL licenses are generally designed as copyleft licenses. This means that users are free to use code under such licenses, but derivative works need to be distributed under the same license. The libUSB library is under the LGPLv2.1 license [49]. This “lesser” GPL license is more permissive since it allows the library to be used even in proprietary software under some conditions. For example, an LGPLv2.1 licensed library can generally be used without disclosing source code when it is distributed separately, or in other words, dynamically linked [50].

MbedTLS is distributed under the Apache 2.0 license. Code licensed with Apache 2.0 can be used freely, even in commercial closed source programs. However, problems may arise when mixing code under different licenses. Due to the licensing requirements, Apache 2.0 code can be included in GPLv3 applications, but not the other way around. Furthermore, Apache 2.0 and GPLv2 are not compatible [51].

The MIT license in cJSON [52] is a permissive license that imposes minimal restrictions on the use of code. It can be used in proprietary or GPL licensed programs, requiring only the license text to be included in the software.

The RemoteHub itself was chosen to be distributed under the GPLv3 license, which means it is open source and free for use outside of Aava Mobile. This license permits the use of the libraries and follows the spirit of the original USB/IP tools. The USB/IP driver files in the kernel source directory folder *drivers/usb/usbip* are licensed under the GPLv2 or any later version.

5.4 RemoteHub usage

The RemoteHub client and server applications are configurable command-line executables but can in the future be extended with a graphical user interface. Initially, the applications were built to only display information about their internal state. This is only informative data and is not needed for operation. The server command line output is shown in Figure 34, and the client interface is presented in Figure 35.

Busid	Manufacturer	Product	Exported
1-1.1	ZTE, Incorporated	ZTE LTE Technologies M	False
1-1.3	AudioQuest	AudioQuest DragonFly B	False
2-1.2.1	Dell	Dell USB Optical Mouse	False
2-1.2.2.1.2	Creative Technology Lt	VF0610 Live! Cam Socialize HD	False
2-1.2.2.3.1	FTDI	FT232R USB UART	False

Figure 34. The RemoteHub server interface shows devices attached to the server.

```
Client started
Attached ZTE, Incorporated - ZTE LTE Technologies MSM at 192.168.1.185:3240
Attached Creative Technology Ltd. - VF0610 Live! Cam Socialize HD at 192.168.1.185:3240
Attached FTDI - FT232R USB UART at 192.168.1.185:3240
```

Figure 35. RemoteHub client interface showing device attach events.

RemoteHub is designed to be used autonomously, meaning that after the initial configuration, there is no need for user input. During initial configuration, the server and client are given a path where TLS certificates are located. Both server and client applications use configuration files to convey the TLS certificate path and other initial information. The configuration files are JSON formatted, and for parsing these files, RemoteHub uses the cJSON library [44]. The initially supported configuration file parameters for client and server are presented in Table 8 and Table 9, respectively, along with a description of each value.

Table 8. RemoteHub client supported configuration file options

Configuration option	Data format	Description
config_version	Number	The configuration file version supported by the application. The initial RemoteHub application supports version 1.
use_tls	Boolean	Whether to use TLS security or unsecured TCP for existing USB/IP application compatibility.
ca_path	String	The path in the VIOBox folder hierarchy to the certification authority certificate that can be used to verify whether a detected server can be trusted, and the connection initialized.

Table 9. RemoteHub server supported configuration file options

Configuration option	Data format	Description
config_version	Number	The configuration file version supported by the application. The initial RemoteHub application supports version 1.
server_name	String	A human-readable name that is assigned to this server. It will be broadcast to the client applications. The client application can then evaluate whether a connection will be attempted.
bcast_enabled	Boolean	Whether to send automatic UDP detection packets or not.
use_tls	Boolean	Whether to use TLS security or plain TCP for existing USB/IP application compatibility.
port	Number	Which network port to set listening for client requests.
cert_path	String	The path to the Privacy Enhanced Mail (PEM) formatted TLS public certificate assigned to this server and associated with the following private key.
key_path	String	The path to the PEM formatted TLS private key of this server.
key_pass	String	The password to use to decrypt the private key if protected.
disable_array	Array of numbers	An array of disabled USB buses that are ignored by the application. Clients cannot use devices connected to them.

The client application is started by first loading the existing USB/IP VHCI host controller driver module with the “*modprobe vhci-hcd*” command and then starting RemoteHub with the command “*rh_client -c <path to configuration file>*”. Likewise, the server startup command is in a similar format “*rh_server -c <path to configuration file>*” but with the server, no kernel modules need to be loaded. After startup, a user can insert a USB device into the server, and it will automatically be usable on the client. All parameters accepted by the created applications are listed in Table 10.

Table 10. RemoteHub supported application parameters

Parameter	Application	Description
-c / --config <path to config file>	Client and Server	Used to pass the configuration file during initialization.
-v / --version	Client and Server	Returns the version of RemoteHub and the versions of the used libraries.
-i / --ip <IPv4 address>	Client	Forces the client to use a specific server at a given IP address.
-p / --port <port number>	Client	Forces the client to use a specific port with a previously given IP address.

Applications can be stopped with a SIGINT signal generated by issuing a Ctrl+C key combination or other means. After catching this signal, they exit controllably, terminating open connections and performing exit cleanup.

5.5 USBIP-win software

The existing USBIP-win client application can be used with VIOBox to support Windows devices because RemoteHub is compliant with the existing USB/IP tool protocol. However, without modifications, only unencrypted transfers are possible. As a proof of concept, a test build of USBIP-win was created with TLS support. The TLS was implemented with the Windows MbedTLS library. However, the automatic discovery was not implemented. The system was functional but needs additional work that is left for the future.

6 RESULTS

After the initial RemoteHub software work was completed, measurements were conducted to evaluate performance and optimize the software. Most of the tests measured data transfer speed by transferring data from a USB 3.0 flash drive. This performance test setup was expected to stress the system greatly and return easily quantifiable results. The usability of a USB web camera and a mouse was also assessed. These devices enabled the testing of all four USB transfer types, which was necessary to uncover possible underlying issues in the software.

RemoteHub was first tested in an optimal environment where only the software performance was measured while keeping the other test variables as constant as possible. Issues that affected performance were thought to at least include network delays and encryption ciphers. From the technical perspective, transfer speed tests were run with the help of a script that automated the testing process. This allowed averaging to be easily added for reliable measurements. During all the performance tests, one gibibyte (GiB) of test data was read from the USB stick, and transfers were averaged five times.

The tests were conducted using the *dd* command presented in Figure 36 that read data from the flash drive to the */dev/null* node to avoid erroneous results due to the use of the client file system and storage medium. Prior to testing, it was discovered that an 8 MiB block size worked well with the test flash drive, and this was used in all tests. System memory caches were also cleared using the command in Figure 37 before every read to avoid the caches affecting performance.

```
dd if=${TEST_FILE_PATH} of=/dev/null bs=8M conv=fdatasync
```

Figure 36. Command that was used to read test files during performance testing.

```
echo 3 > /proc/sys/vm/drop_caches
```

Figure 37. Command that was used to clear memory caches.

The first tests were run in a loopback environment, where no data was transferred over the network and server and client were run on the same computer. Following that, the focus was shifted to VIOBox hardware. The VIOBox hardware tests were carried out on both the Raspberry Pi 3 Model B and the Orange Pi Zero SBCs with either an x86 computer or an Aava tablet as a client. These tests were performed to evaluate overall performance and to find the optimal encryption ciphers for use. Finally, the actual VIOBox system was tested in real-life scenarios with an Aava tablet client connected to it through a wireless network.

6.1 Optimal environment

The RemoteHub and the existing USB/IP tools were measured in an optimal environment, and a setup was created where software comparisons could be reliable. The computer used in these tests was an HP EliteBook 8570w with an Intel i7-3840QM processor and Manjaro Linux as the operating system. The network-related influence was mitigated by running both the server and client on the same computer using a localhost loopback interface with no communication over network cables. This setup was ideal for isolating the effects of network delays, which were known to have a severe impact on performance. The network Round-Trip Time (RTT) was measured with the *ping* command to be 0.083 ms in an average of 100 packets on localhost. Also, a local copy test was performed for reference. The local copy was a baseline that represented the absolute maximum transfer speed with a standard USB connection. Then, a 1 GiB test file with either random data or only the value zero was transferred from the Kingston USB 3.0 stick into the `/dev/null` device node. The results are presented in Table 11.

Table 11. Transfer rates measured from a Kingston USB 3.0 flash drive (unless otherwise specified, connected to a USB 3.0 port on the server computer)

Test setup	Test data	Avg transfer speed (5 reads)
Local copy	1 GiB of random	95.93 MiB/s
Local copy	1 GiB of zero	95.76 MiB/s
Existing USB/IP tools	1 GiB of random	71.02 MiB/s
Existing USB/IP tools	1 GiB of zero	72.96 MiB/s
Unencrypted RemoteHub	1 GiB of random	62.57 MiB/s
Unencrypted RemoteHub	1 GiB of zero	64.11 MiB/s
Unencrypted RemoteHub (USB 2.0 port)	1 GiB of random	25.90 MiB/s
Unencrypted RemoteHub (USB 2.0 port)	1 GiB of zero	25.84 MiB/s

Following the performance comparisons, TLS performance was evaluated to see how encryption affected the speed of RemoteHub. In these tests, only 1 GiB random data files were used. During initial testing, it was found that the RemoteHub TLS libraries were compiled with debug settings, reducing cryptographic performance. After this issue was fixed, the cipher performance is presented in Figure 38. Although a full TLS cipher suite contains additional ciphers, only the encryption and verification ciphers are shown in Figure 38. Other ciphers would not have contributed to the performance since the test was conducted after a connection had been established. However, the used key exchange and authentication ciphers were DHE-RSA in all cases.

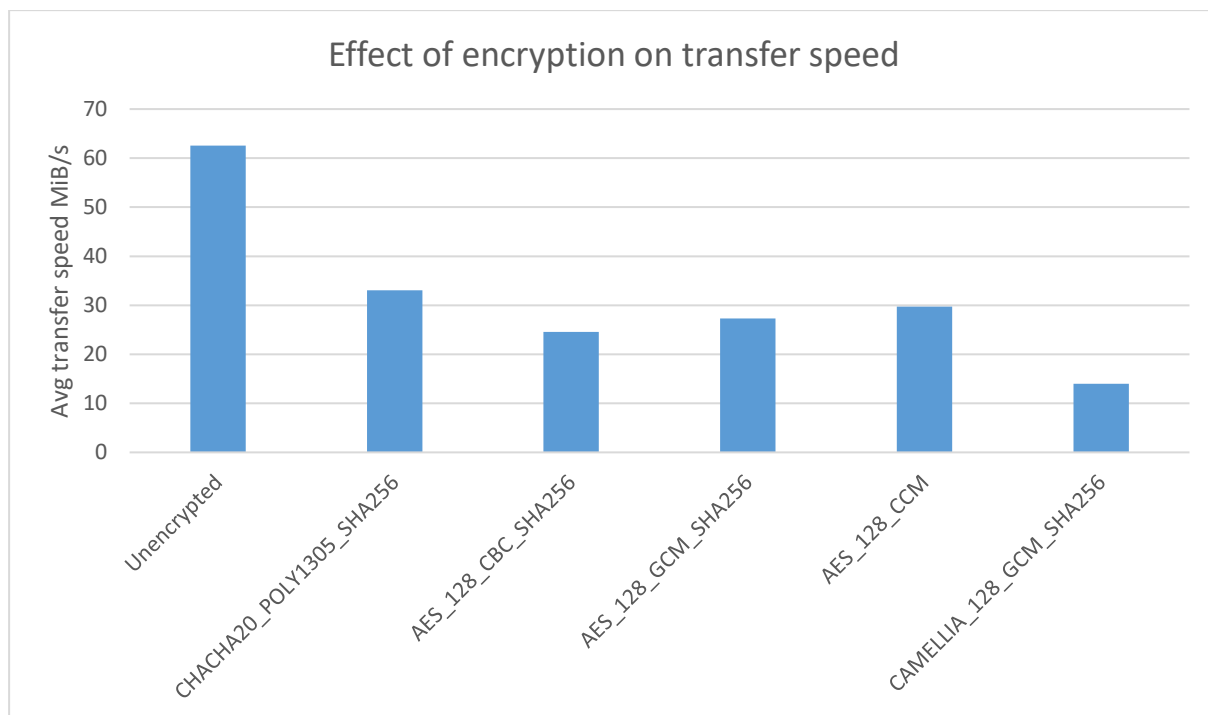


Figure 38. Effect of cipher suite data encryption and verification ciphers on locally used RemoteHub, where 1 GiB of random data was read from a Kingston flash drive connected to a USB 3.0 port.

During encryption cipher tests, it was seen that CPU use was correlated with the transfer speed. Processor use was the highest with CAMELLIA_128_GCM_SHA256 and the lowest with CHACHA20_POLY_1305_SHA256.

Next, the effect of network delay was also useful to be measured in the optimal environment because it can be synthetically generated and controlled. The delay was generated using the command shown in Figure 39.

```
tc qdisc add dev lo root netem delay <delay value>
```

Figure 39. The command that was used to add delay to the socket communication.

The used delay was set to half of the desired RTT value since the delay shaping affected both server and client applications. The delay was verified with the *ping* command. These tests were conducted with an unencrypted RemoteHub setup, and the results are presented in Figure 40.

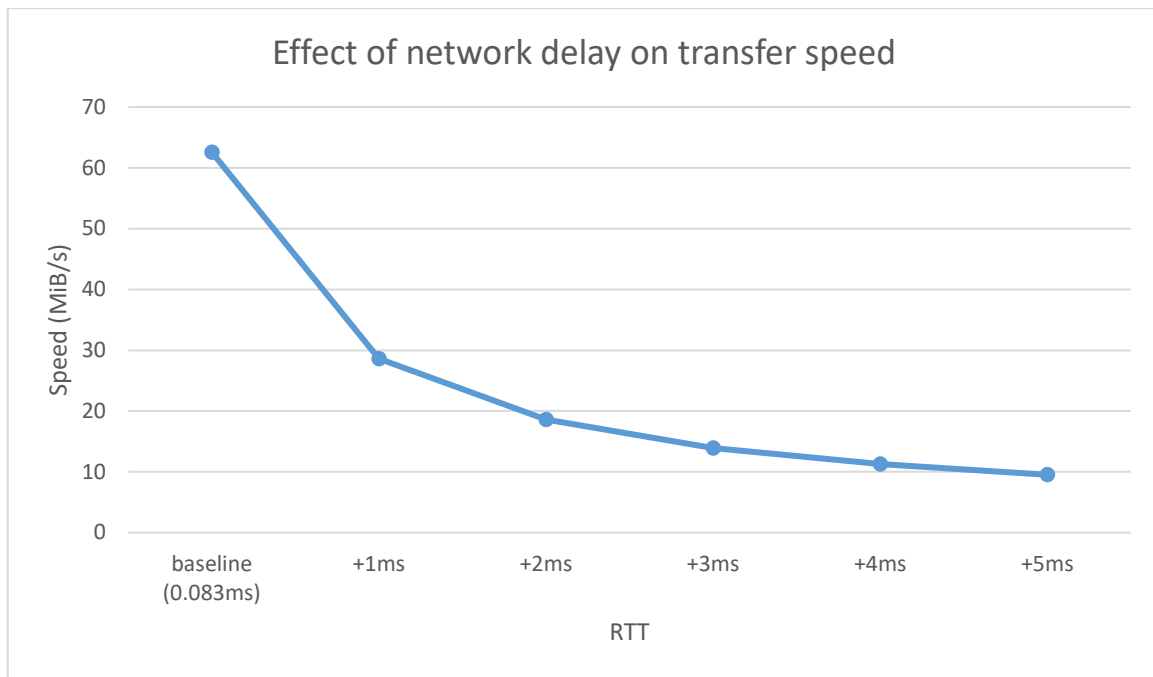


Figure 40. The RTT between client and server on the same computer and the effect on transfer speed.

Following these tests, a Web camera and mouse were tested in an unencrypted environment. First, a Logitech MX518 mouse was used. It has an HID report polling interval (*bInterval*) of 10, which indicates a period of 10 ms. However, due to rounding in the kernel, it was actually read at 8 ms intervals (125Hz) [53]. During testing, starting at a delay of 20 ms, the mouse movement visibly started to slow down. A delay of 150 ms or more could be considered completely unusable. A Microsoft LifeCam HD-5000 was tested with the *Webcamoid* web camera application. It was seen that a 640x480 30fps preview stream was functional up until a delay of approximately 10 ms when the preview stopped functioning. The preview was still working with a delay of 8 ms.

6.2 VIOBox device

First, VIOBox tests were performed to find the optimal encryption ciphers for use. These tests were like the transfer speed tests conducted previously. However, this time, the VIOBox server and an Intel i7-2600-based x86 client computer were connected by a wired network connection. Transfer speeds using different encryption and verification cipher suites are presented in the following Figure 41. These are the same suites that were used in the optimal environment.

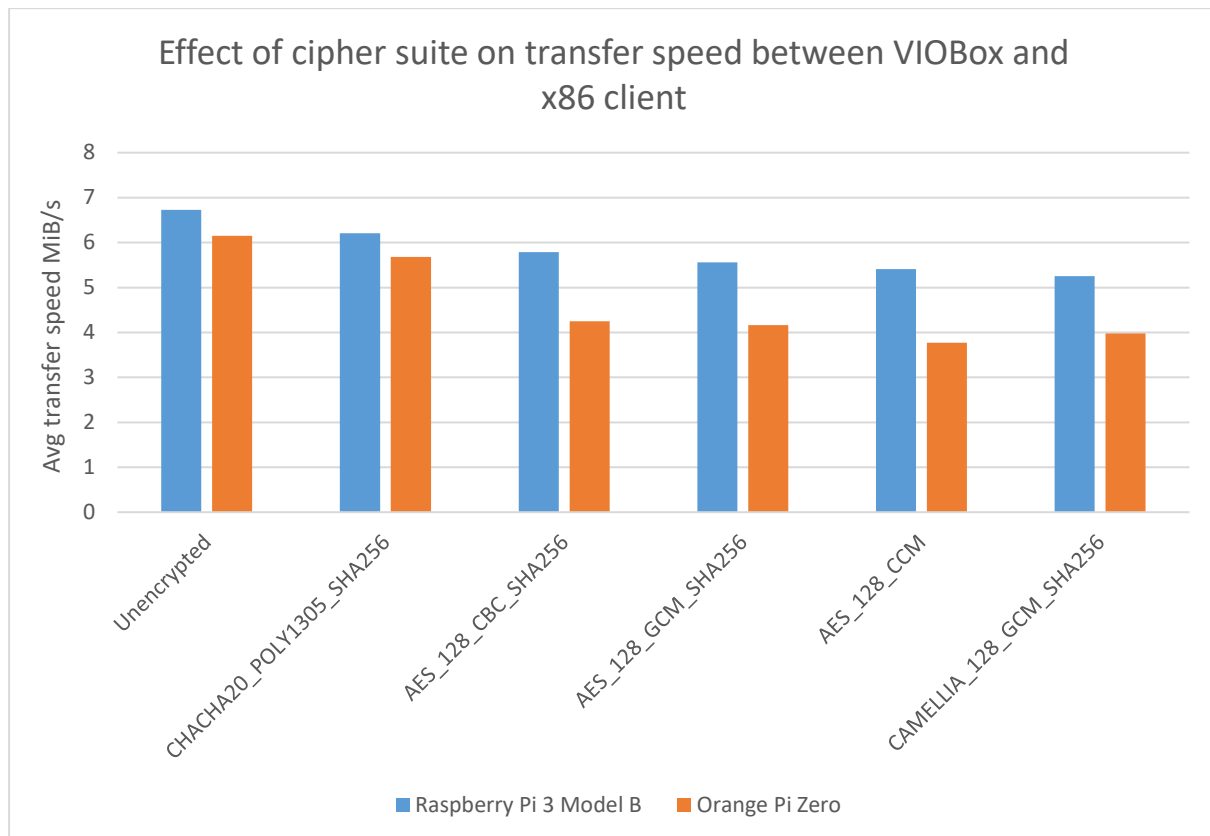


Figure 41. The impact of data encryption and verification ciphers on the transfer speed between the VIOBox server and an x86 computer client using RemoteHub.

The *ping* command reported an average RTT of 0.411 ms between the client and Raspberry Pi 3 Model B and, likewise, 0.240 ms between the client and Orange Pi Zero on an average of 100 tests. The used network cables were gigabit capable, and the same on both tests.

In real-life performance tests, the Orange Pi Zero-based VIOBox was used with an Aava x86-based tablet client with an Intel Atom E3940 processor. The most suitable cipher suite, TLS_DHE_RSA_WITH_CHACHA20_POLY1305_SHA256, was selected and used during the tests. The VIOBox was connected to the network using an ethernet cable, but the tablet used a Wireless Local Area Network (WLAN) connection. Two distances were tested, with a separation of 50 cm and 4 m between the router and tablet. The tablet also had an unobstructed line of sight to the router. As a note, the exact fine positioning of the device was observed to affect the RTT. Transfer speeds and RTT values are shown in Table 12.

Table 12. Transfer speed and observed delay with Aava tablet as client and Orange Pi VIOBox as a server

Distance	Transfer speed	RTT (Avg. of 100)	RTT std. dev.
50 cm	3.20 MiB/s	2.67 ms	0.75 ms
4 m	2.71 MiB/s	3.65 ms	2.29 ms

More testing was performed in the 50 cm range to assess the impact on input devices. A Logitech MX518 mouse was tested with the *evhz* tool [54]. It was seen that on average the mouse was returning data at 125 Hz, the rate being occasionally higher and lower than this

value. The Microsoft LifeCam HD-5000 web camera was tested with the V4L2 test benchmark tool *qv4l2* [55]. It was seen that the video was only suitable to be used at a maximum 424x240 resolution. Above this, the video was either not working or frame dropping was too severe for use. The camera was set to output data video at 30 fps, and it was verified that the frame rate was 29.98 when connected to the tablet directly. Using VIOBox, occasional dropped frames were present at that resolution, and the frame rate was around 19 fps in the same conditions as the reference test. It was seen that even with a minimum resolution of 160x120, the average frame rate was also at around 19 fps.

RemoteHub-related requirements were also verified during these tests. This especially meant testing three USB devices simultaneously and ensuring that the hot-plugging delay was under 10 seconds. Both requirements were supported.

7. DISCUSSION

Part of this thesis was to evaluate how existing open-source USB/IP software components could be used or extended to implement data security and automatic use features. The presented solution was to create new RemoteHub USB sharing tools. RemoteHub was built to use TLS for data security and UDP broadcast packets for automatic discovery. It leverages the USB/IP tool protocol and the existing USB/IP virtual host controller that is included in most Linux distributions. This also helps with future development, since there is a Windows implementation that could be updated with the required features in the future. As a part of the software work, the USB/IP server was built in user space. This enables better maintainability and portability, but also imposes a small performance penalty on the operation.

After the software work was concluded, the RemoteHub software and the VIOBox accessory performance were measured and optimized. The main goal of this testing was to find the main performance bottlenecks in the prototype VIOBox system for future reference. Most tests were conducted using a script that read data from a physical USB 3.0 flash drive, simulating USB mass storage use cases. This testing returned quantifiable results and was used for benchmarking. Additionally, a web camera and input devices were assessed at a more general and subjective level. Testing was conducted both on a theoretical optimal simulation environment and on VIOBox hardware.

First, RemoteHub was tested in optimal conditions to isolate network delay. During these tests, both server and client applications were used on the same computer using the localhost loopback interface. It was verified that the newly created implementation had some negative performance impact. Remotehub with the user space server was approximately 12 % slower than existing Linux USB/IP tools which operate in kernel space. The main issue is likely the data copying between kernel and user space, which was identified by Hirofuchi in his research. However, the execution logic of RemoteHub and libUSB may also add overhead.

The addition of data security features imposed additional penalties depending on the hardware and used cipher suite. Optimal environment performance with optimal encryption ciphers was approximately half of that without encryption. Luckily, this did not fully translate to VIOBox. When optimizing VIOBox encryption ciphers, it was seen that on the selected Orange Pi Zero hardware, this resulted in approximately 8 % slower performance with the most optimal `TLS_DHE_RSA_WITH_CHACHA20_POLY1305_SHA256` cipher suite compared to being unencrypted. It is also important to note the varying security implications of the tested cipher suites. However, at the time of testing, the optimal cipher suite was also classified as secure [56].

Network delays introduced significant performance degradation. This can be especially problematic with VIOBox since it uses a wireless network connection, which introduced more delay compared to a wired connection. The measured RTT values were also more unpredictable compared to a wired connection. The issue of network latency was also discovered in the original USB/IP research [6]. The VIOBox data transfer performance degradation with increasing latency can likely be attributed to the client USB drivers being designed to work in conventional, minimal latency USB environments. Performance is affected if a client needs to wait for an URB to be completed before a new one is issued. The performance degradation due to encryption can likely be attributed to delay caused by a higher processor workload.

From the VIOBox hardware perspective, the network adapters in used SBCs were only capable of operating at a maximum speed of 100 Mbps and the USB ports were only capable of USB 2.0. The performance of VIOBox could likely be improved by changing the used hardware. However, as seen in the optimal environment, the improvements may be limited

due to latency in the network, which sets a ceiling for transfer speed and can be hard to control on user premises.

The performance of the RemoteHub software was deemed acceptable in the intended use case. The use of lower-performance hardware can be accepted as a cost-saving measure. As was determined during testing, input devices should not be notably impacted by the VIOBox system due to the relaxed polling intervals in USB interrupt transfers. However, VIOBox still needs further development before it is ready for commercial use. For instance, the delivery mechanism for the initial data has not been decided. Some options include, for example, reading it from an NFC tag, a USB flash drive, or using SSH / Secure Copy (SCP) methods during provisioning. Proper services or startup scripts should also be created so that both the client and server can automatically begin execution after startup.

The RemoteHub software has many aspects that can be developed further, and some existing choices also worked well. Following the prototype phase, Android and Windows support could be added, so that all Aava devices are supported. A GUI using which the system could be managed would be nice to have but not necessary. It was seen in an optimal environment that the content of transferred data had a negligible effect on transfer speed. To speed up VIOBox operation, data compression could be added to attempt to boost performance. The compression likely hurts performance on fast networks and optimal hardware but could possibly be of help on VIOBox. Also, more fine-grained access controls could be added. The prototype used TLS certificates themselves for rudimentary access control. This was possible since the system was designed to use all the devices on the server. Perhaps in the future, VIOBox could also be offered in other forms, such as a larger server that could serve multiple clients. Then, controlling access would become more relevant. One internal improvement would be to lift more functionality to the application level so that the libraries stay as lean and flexible as possible. There are also other smaller improvements to be made, some of which have undoubtedly not yet been uncovered. Due to this, user experience testing would also be beneficial. In the current design, asynchronous callbacks worked well for handling the application logic. This enabled, for example, the client to connect immediately and automatically to a server after receiving a notification.

8. SUMMARY

This thesis documented the software development process for an Aava Mobile VIOBox USB remote use prototype accessory, which was designed to serve as a remote wireless USB hub for Aava tablets. The existing open-source software USB remote use tools were discovered to be insufficient for VIOBox as is. VIOBox was required to support secure data transfers and function without user intervention. Therefore, after evaluation of options, new RemoteHub tools were decided to be created. These tools consist of server and client applications for Linux-based operating systems and use the existing USB/IP protocol and virtual host controller on the client.

This thesis began with the evaluation of existing commercial and open-source USB remote use solutions. Open-source USB/IP tool features and limitations were explored in more detail, and some ways to overcome the key limitations were presented. The evaluation was followed by a review of the USB and USB/IP technical background. This was necessary as preparation for creating the new RemoteHub system. After the review of theory, the RemoteHub tool was introduced. The RemoteHub requirements, design choices, and internal details were also presented, along with a brief overview of the VIOBox device hardware and mechanics.

Finally, the RemoteHub software and VIOBox device were benchmarked. First, measurements verified that network delay had a great effect on performance. It was seen that in an optimal environment, even one millisecond of delay can lead to a halving of the data transfer speed when reading data from a USB stick. Delay also originated from the encryption implementation. The encryption calculations also halved the transfer speed, even with the most favorable ciphers in an optimal environment.

It was also seen that the selected VIOBox HW components were limited in performance aspects. Due to higher latency, slower network adapters, and the lack of USB 3.0 ports, the performance was approximately 10 % of the optimal environment. It was found that both the evaluated Orange Pi Zero and the Raspberry Pi 3 Model B SBCs had similar performance, with the Raspberry Pi being slightly faster. Further assessing the actual use case with an Aava tablet and wireless network connection, the performance was again roughly half of the speed achievable with a wired connection. The data transfer speed that was achieved when reading data from a USB stick with the Orange Pi Zero VIOBox was approximately 3 MiB/s from 50 cm of the access point using the optimal encryption ciphers. However, the performance issues were determined not to be critical for the VIOBox use case. It is acceptable for the occasional flash drive data transfer to be less performant. The VIOBox is mainly dedicated to human interface devices that were seen to tolerate real-life network conditions well. Devices such as keyboards and mice were not seen to be meaningfully affected when using the VIOBox.

9. REFERENCES

- [1] Universal Serial Bus Specification Revision 2.0 (2000). Compaq Computer Corporation, Hewlett-Packard Company, Intel Corporation, Lucent Technologies Inc, Microsoft Corporation, NEC Corporation, Koninklijke Philips Electronics N.V., 622 p.
- [2] Hirofuchi, T., Kawai, E., Fujikawa, K. & Sunahara H. (2005) USB/IP—A Peripheral Bus Extension for Device Sharing over IP Network. In: The Proceedings of the FREENIX Track: USENIX Annual Technical Conference, p. 47–60.
- [3] Digi International and Inside Out Networks (read 3.5.2022) Inside Out Networks Revolutionizes Remote Peripheral Connectivity with Debut of USB Over IP™ Technology. URL: <https://web.archive.org/web/20020603232319/http://www.digi.com/corporateinfo/news/newsreleases/111201.html>
- [4] Inside Out Networks (read 3.5.2022) Remote USB Over IP Concentrator Preliminary Specs. URL: <https://web.archive.org/web/20030421060700/http://www.ionetworks.com/products/anywhereusb.pdf>
- [5] Hirofuchi T. (2004) A design of device control and sharing system over IP network and its implementation. Master's thesis. Nara Institute of Science and Technology, Graduate School of Information Science, Department of Information Systems.
- [6] Hirofuchi T. (2007) USB/IP: universal serial bus extension over IP network. Doctoral dissertation. Nara Institute of Science and Technology, Graduate School of Information Science, Department of Information Systems.
- [7] Hirofuchi T. (read 22.5.2022) Staging: USB/IP: add common functions needed. URL: <https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=05a1f28e879e3b4d6a9c08e30b1898943f77b6e7>
- [8] Hirofuchi T. (read 22.5.2022) staging: usbip: add userspace code. URL: <https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=0945b4fe3f016900f1f68255e24920b28624a9aa>
- [9] Torvalds L. (read 22.5.2022) Merge tag 'usb-3.17-rc3'. of [git://git.kernel.org/pub/scm/linux/kernel/git/gregkh/usb](https://git.kernel.org/pub/scm/linux/kernel/git/gregkh/usb). URL: <https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=848298c6fb36fbe459854e376ce90af32ba6e1ce>
- [10] Du Y. (read 22.5.2022) usbip: vhci-hcd: Add USB3 SuperSpeed support. URL: <https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=1c9de5bf428612458427943b724bea51abde520a>

- [11] lepton-wu (read 22.5.2022) usbip and vbus driver for windows. URL: <https://sourceforge.net/p/usbip/git-windows/ci/6e9ddd379dd0a9f1123bbe095c4a634bca2a59e7>
- [12] Digi international (read 22.5.2022) AnywhereUSB® Plus User Guide. URL: <https://www.digi.com/resources/documentation/digidocs/90002383/default.htm>
- [13] Solid State Supplies Ltd (read 28.5.2022) Digi AnywhereUSB Plus. URL: <https://www.sssltd.com/product/digi-anywhereusb-plus/>
- [14] Coolgear Inc (read 22.5.2022) AnyplaceUSB-S2 USB SERVER USER'S MANUAL. URL: <https://www.coolgear.com/wp-content/uploads/2020/01/AnyplaceUSB-S2-Manual.pdf>
- [15] USBGear.com (read 28.5.2022) 2-Port USB over Ethernet USB Device Server. URL: <https://www.usbgear.com/AnyplaceUSB-S2.html>
- [16] VirtualHere Pty. Ltd. (read 22.5.2022) VirtualHere EasyFind FAQ. URL: https://www.virtualhere.com/easyfind_faq
- [17] VirtualHere Pty. Ltd. (read 22.5.2022) VirtualHere website. URL: <https://www.virtualhere.com/>
- [18] Electronic Team, Inc (read 22.5.2022) USB Network Gate. URL: <https://www.net-usb.com/>
- [19] Cho K. (read 22.5.2022) USB/IP for Windows. URL: <https://github.com/cezanne/usbip-win>
- [20] Microsoft Corporation (read 22.5.2022) Driver Signing Policy. URL: <https://docs.microsoft.com/en-us/windows-hardware/drivers/install/kernel-mode-code-signing-policy--windows-vista-and-later->
- [21] Samonas S. (2014) THE CIA STRIKES BACK: REDEFINING CONFIDENTIALITY, INTEGRITY AND AVAILABILITY IN SECURITY. Journal of Information System Security (JISSec), Volume 10, Issue 3, p. 21-45
- [22] SSL.com (read 22.5.2022) What Is a Certificate Authority (CA)? URL: <https://www.ssl.com/faqs/what-is-a-certificate-authority/>
- [23] Computerphile (read 22.5.2022) TLS Handshake Explained. URL: <https://youtu.be/86cQJ0MMses>
- [24] Driscoll M. (read 22.5.2022) The Illustrated TLS 1.2 Connection. URL: <https://tls12.ulfheim.net/>

- [25] Keyfactor (read 4.6.2022) An Introduction to Cipher Suites. URL: <https://www.keyfactor.com/blog/cipher-suites-explained/#:~:text=Cipher%20suites%20are%20sets%20of,communications%20between%20clients%20and%20servers.>
- [26] Microsoft Corporation (read 22.5.2022) USB endpoints and their pipes. URL: <https://docs.microsoft.com/en-us/windows-hardware/drivers/usbcon/usb-endpoints-and-their-pipes#:~:text=those%20two%20terms.-,USB%20endpoint,into%20control%20and%20data%20endpoints>
- [27] Future Technology Devices International Ltd. (read 22.5.2022) Technical Note TN_113 - Simplified Description of USB Device Enumeration. URL: https://ftdichip.com/wp-content/uploads/2020/08/TN_113_Simplified-Description-of-USB-Device-Enumeration.pdf
- [28] Corbet J., Rubini A. & Kroah-Hartman G. (2005) Linux Device Drivers, Third Edition, 615 p.
- [29] Microchip Technology Incorporated (read 22.5.2022) USB Transfer Types. URL: <https://microchipdeveloper.com/usb:transfer>
- [30] Kroah-Hartman G. (read 22.5.2022) Writing USB Device Drivers. URL: https://www.kernel.org/doc/html/latest/driver-api/usb/writing_usb_driver.html
- [31] Venkateswaran S. (2008) Essential Linux Device Drivers, 744 p.
- [32] The kernel development community (read 22.5.2022) Platform Devices and Drivers. URL: <https://www.kernel.org/doc/html/latest/driver-api/usb/usb.html#usb-host-side-api-model>
- [33] The libUSB project (read 22.5.2022) libUSB wiki. URL: <https://github.com/libusb/libusb/wiki>
- [34] The kernel development community (read 22.5.2022) The Linux kernel v5.17.9 source tree file hcd.c. URL: <https://elixir.bootlin.com/linux/latest/source/drivers/usb/core/hcd.c>
- [35] The kernel development community (read 4.6.2022) USB/IP protocol. URL: https://www.kernel.org/doc/html/latest/usb/usbip_protocol.html
- [36] Nagle J. (1984) RFC 896 - Congestion Control in IP/TCP Internetworks
- [37] The Raspberry Pi Foundation (read 22.5.2022) Raspberry Pi 3 Model B. URL: <https://www.raspberrypi.com/products/raspberry-pi-3-model-b/>

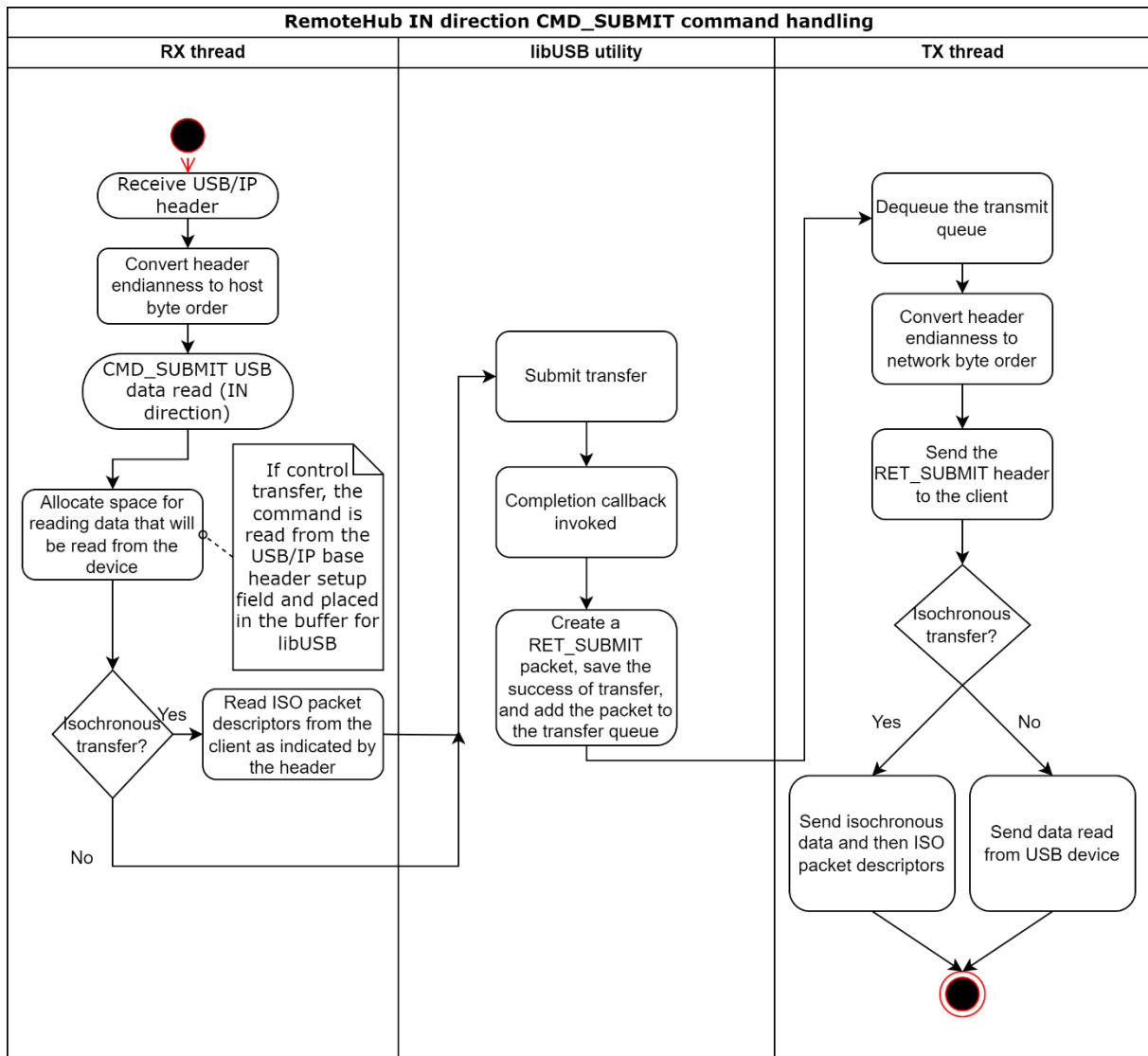
- [38] Shenzhen Xunlong Software CO., Limited (read 22.5.2022) What's Orange Pi Zero? URL: <http://www.orangepi.org/orangepizero/>
- [39] Armbian forum (read 22.5.2022) crypto engine (openvpn related, aes-ni). URL: <https://forum.armbian.com/topic/2099-crypto-engine-openvpn-related-aes-ni/>
- [40] The Raspberry Pi Foundation (read 22.5.2022) Raspberry Pi OS (64-bit). URL: <https://www.raspberrypi.com/news/raspberry-pi-os-64-bit>
- [41] Zhou J.Y (2004) Functional Requirements and Non-functional requirements, Concordia university, Montreal, Quebec, Canada
- [42] Bradner S. (1997) RFC2119 - Key words for use in RFCs to Indicate Requirement Levels
- [43] TrustedFirmware.org (read 22.5.2022) MbedTLS info page. URL: <https://www.trustedfirmware.org/projects/mbed-tls/>
- [44] Laitinen J. (read 22.5.2022) RemoteHub Git repository. URL: <https://github.com/Prototyyppe/RemoteHub>
- [45] Amazon.com, Inc. (read 22.5.2022) Pub/Sub Messaging - Asynchronous event notifications. URL: <https://aws.amazon.com/pub-sub-messaging/>
- [46] The kernel development community (read 22.5.2022) Linux kernel coding style. URL: <https://www.kernel.org/doc/html/latest/process/coding-style.html>
- [47] Preston-Werner T. (read 22.5.2022) Semantic Versioning 2.0.0. URL: <https://semver.org/>
- [48] Kitware, Inc and Contributors (read 5.6.2022) CMake Reference Documentation. URL: <https://cmake.org/cmake/help/latest/index.html>
- [49] The libUSB project (read 22.5.2022) libUSB license. URL: <https://github.com/libusb/libusb/blob/master/COPYING>
- [50] The Free Software Foundation (read 22.5.2022) Frequently Asked Questions about the GNU Licenses. URL: <https://www.gnu.org/licenses/gpl-faq.en.html#LGPLStaticVsDynamic>
- [51] The Apache Software Foundation (read 22.5.2022) APACHE LICENSE V2.0 AND GPL COMPATIBILITY. URL: <https://www.apache.org/licenses/GPL-compatibility.html>
- [52] Gamble D. and cJSON contributors (read 22.5.2022) cJSON license. URL: <https://github.com/DaveGamble/cJSON#license>

- [53] Archwiki (read 29.5.2022) Mouse polling rate. URL: https://wiki.archlinux.org/title/mouse_polling_rate
- [54] Kelling I. (read 29.5.2022) evhz git repository. URL: <https://git.sr.ht/~iank/evhz>
- [55] Ubuntu manuals (read 29.5.2022) qv4l2 - A test bench application for video4linux devices. URL: <http://manpages.ubuntu.com/manpages/impish/man1/qv4l2.1.html>
- [56] ciphersuite.info – Rudolph H.C & Grundmann N. (read 22.5.2022) TLS_DHE_RSA_WITH_CHACHA20_POLY1305_SHA256 evaluation result. URL: https://ciphersuite.info/cs/TLS_DHE_RSA_WITH_CHACHA20_POLY1305_SHA256/

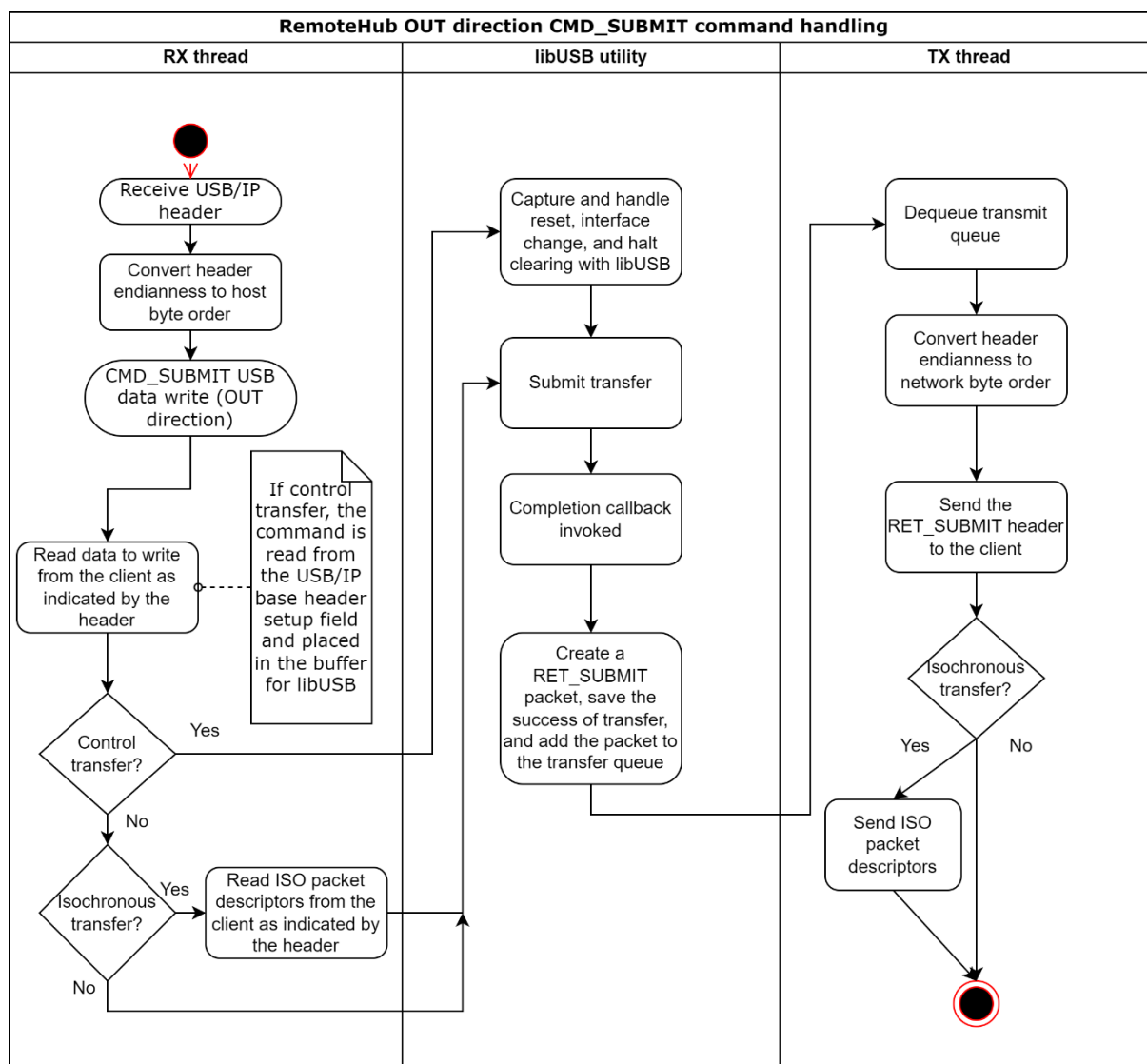
10.APPENDICES

- Appendix 1 The handling of IN direction USBIP_CMD_SUBMIT in RemoteHub server
- Appendix 2 The handling of OUT direction USBIP_CMD_SUBMIT in RemoteHub server
- Appendix 3 The handling of USBIP_CMD_UNLINK in RemoteHub server

Appendix 1 The handling of IN direction USBIP_CMD_SUBMIT in RemoteHub server



Appendix 2 The handling of OUT direction USBIP_CMD_SUBMIT in RemoteHub server



Appendix 3 The handling of USBIP_CMD_UNLINK in RemoteHub server

