Aalto University
School of Electrical
Engineering

Master’s programme in Automation and Electrical Engineering

High Speed Ethernet to USB Interface for Embedded System Communication

Thales Mendes Sampaio

Master’s Thesis
2023

Aalto University
School of Electrical Engineering

Author Thales Mendes Sampaio

Title High Speed Ethernet to USB Interface for Embedded System Communication

Degree programme Automation and Electrical Engineering

Major Control, Robotics and Autonomous Systems

Supervisor Prof. Jukka Manner

Advisor David Muñoz Martinez (MSc)

Collaborative partner GE HealthCare

Date 8 May 2023 **Number of pages** 58+24 **Language** English

Abstract

The producer of patient monitoring systems GE HealthCare has had the need to provide a wired IP communication interface for one of its monitoring USB devices. This was done by using the embedded micro-controller on the charging station of the monitoring device as a data interface between USB and Ethernet. The software solution was designed to have support for Communication Device Classes ACM, ECM, and EEM, for both USB full-speed and high-speed settings.

The software developed was tested using a variety of tools to measure the network performance of the data path, including throughput, latency and packet loss measurements for different packet pattern configurations. The functional and non-functional requirements for the charging device were met, as it was able to provide the USB monitoring device speeds of 94 Mbps through the 10/100 Mbps Ethernet interface, with minimal packet losses, as well as sustaining a stable connection under high load scenarios.

Keywords USB, Ethernet models, real-time systems, embedded systems

Preface

This thesis was written during my employment in GE HealthCare, working with the ambulatory monitoring system Portrait Mobile. Throughout this project I had the opportunity to explore various aspects of embedded systems, communication protocols, and software engineering. I would like to express my sincere gratitude to my advisor David Muñoz for the chance to have this part of the Portrait Mobile development as my master's thesis topic. A special thank you to my colleagues who have provided guidance and support. I also thank Professor Jukka Manner for his assistance on the writing process and thesis structure. Lastly, I extend my appreciation to my family and friends for their support, understanding and encouragement. Their belief in my abilities has been a constant source of motivation.

Helsinki, 5 May 2023

Thales M. S. Engineer

Contents

Abbreviations and Acronyms

ACM	Abstract Control Model
API	Application Programming Interface
CDC	Communication Device Class
CRC	Cyclic Redundancy Check
DMA	Direct Memory Access
DPRAM	Dual-port Random Access Memory
ECM	Ethernet Control Model
EEM	Ethernet Emulation Model
EMA	European Medicines Agency
FDA	Food and Drug Administration
FIFO	First In First Out
FTDI	Future Technology Devices International
HAL	Hardware Abstraction Layer
IC	Integrated Circuit
ICMP	Internet Control Message Protocol
IP	Internet Protocol
JTAG	Joint Test Action Group
LAN	Local Area Network
LPDDR3	Low-Power Double Data Rate version 3
MAC	Media Access Control
MCU	Micro-controller Unit
PCB	Printed Circuit Board
SOF	Start of Frame
SpO2	Oxygen Saturation
SDRAM	Synchronous Dynamic Random Access Memory
SRAM	Static Random Access Memory
SSH	Secure Shell Protocol
TCP	Transmission Control Protocol
UART	Universal Asynchronous Receiver Transmitter
UDP	User Datagram Protocol
UHI	Universal Serial Bus Host Interface
UHC	Universal Serial Bus Host Controller
USB	Universal Serial Bus
USB-IF	Universal Serial Bus - Implementers Forum
WLAN	Wireless Local Area Network

1 Introduction

A common practice in hospitals is to have devices monitoring the vitals signs of the patients admitted. This is important to increase the safety of the patients before, during and after treatment. Inadequate patient monitoring, inability to respond promptly, and poor patient tracking are examples of medical errors that can cause serious damage to the person being treated [1]. In order to avoid such adverse events, a well trained staff, as well as the use of reliable monitoring equipment can help to give warnings of early deterioration on the patient's health and provide real-time data to help doctors prioritize their attention. Hospitalized patients might experiences complications while recovering from medical procedures or while being observed from a disease process. A delay or failure in recognizing such complications can lead to fatal outcomes [2]. This event, called failure to rescue (FTR), can be mitigated by continuous patient monitoring systems, and it is a key indicator for healthcare quality in hospitals and care units [3].

The client commissioning this thesis, GE HealthCare, is a lead producer of patient monitoring solutions. Their recent innovation on wearable technology includes a series of lightweight wireless sensors the patient wears to have their vital signs monitored. This would provide the patients more physical freedom to move in the hospital's premises while still being monitored. The wearable monitoring devices, would not only provide the patients a feeling of well-being for allowing them to move freely, but also increase their safety by continuously monitoring and giving indications in case of early deterioration, thus reducing the chances of an FTR event [4].

The product proposed by GE HealthCare consists of a set including parameters sensors that acquire data such as blood oxygenation, heart rate, and respiration rate; a visualization device, to manage patients and display data from the monitoring sensors; removable batteries, for powering the sensors; and a docking station, to provide charging and communication capabilities. Throughout this thesis the visualization device will be referred as Hub, and the docking station as Charger. A network service would be available on the hospitals for connecting the Hubs, providing a larger scale monitoring solution and software updates. The Hub functions as the system integrator, connecting to the various sensors attached to the patient and displaying the data to nurses and doctors. Currently, the Hub communicates with the network service via Wireless Local Area Network (WLAN), which is an important feature for when the patient is relocating or moving outside the ward. However, in regular use cases, the Hub will be docked to the Charger for the majority of time, hence a fast and stable wired network connection would be highly desirable, leading to the research question of this thesis: How to provide the Charger a fast USB to Ethernet interface the Hub could use as an alternative communication interface?

A common approach to provide Ethernet connectivity on Universal Serial Bus (USB) devices is to use Communication Device Class (CDC) protocols, which encapsulate data into specific formats, allowing a device and host to communicate [5]. An example usage of these protocols are Ethernet to USB adapters. Although devices like these can easily be found as off-the-shelf products, due to the proprietary USB connector on the Hub and the required functionality of the Charger, this is not an approach that can be taken. To address the demand for wired connectivity, the Charger's software will be redesigned to implement Ethernet emulation CDC protocols and a feasibility study on the performance will be conducted to determine if the solution is appropriate.

This product belongs to the Class II medical device category, thus it must go through a series of approvals from regulatory agencies such as the Food and Drug Administration (FDA) and the European Medicines Agency (EMA) [6]. Having a Local Area Network (LAN) connection on the Charger is an important feature for assisting in the approval for market usage, as it provides the Hub an alternative way to connect to the service network, mitigating the chances of network problems, ultimately increasing the patient's safety. Moreover, the LAN connectivity will provide more bandwidth for future developments on the sensors. With the addition of other sensor types as well as improvements on accuracy and resolution, more data would need to be sent by the Hub. The wired connection would be able to support these future updates.

Although the wearable monitoring system purpose is to enable patients to ambulate, help with comfort and accelerate recovery, for the vast majority of the time, the patient will be in their room and the Hub will be stationary. Using LAN for the Hub connectivity can help to reduce possible wireless interference between other Hubs that are being carried by patients on the move.

The aim of this thesis is to document the software design process for implementing LAN connectivity on the Charger, in order to expand its capabilities to offer "over-the-cable" software updates, as well as provide the Hub a stable and fast way of connecting to a network. To achieve this objective, a technical background study was conducted on Ethernet-style networking over USB. The hardware and micro-controller units involved were analyzed to provide an understanding of their capabilities. Finally, a series of tests and measurements were performed in order to verify the functionality and performance of the LAN connectivity.

In summary, the implementation of the USB to Ethernet interface on the Charger was successful, allowing the Hub to access local area networks using the wired connection. The tests on stability and performance measurements concluded the Charger was able to provide the necessary data bandwidth for future iterations of GE HealthCare's mobile monitoring products.

The remainder of this work is organized as follows. Chapter 2 reviews the literature on USB standards, describing the technology and how Ethernet emulation over USB works. In Chapter 3, the case study is presented and a solution to the LAN connectivity problem is proposed, including the methods used to achieve that. Chapter 4 presents the software design development and describes how the research question was solved. Chapter 5 outlines how the solution was tested and analyses the results. Chapter 6 summarizes the thesis by discussing practical implications and limitations of the study as well as by suggesting further research on the topic.

2 Introduction to USB

This chapter introduces the USB standards and describes the various components related to Ethernet emulation over a USB connection. An overview of how the USB hierarchy works is presented, including details on the physical layer and protocol definitions. Finally, the enumeration steps of a generic USB device will be explained, and the USB Communication Device Class models relevant to this thesis will be introduced.

2.1 Overview

USB, which stands for Universal Serial Bus, is a standard connection type for computer peripherals. It was originally developed in the mid-1990s by a consortium of seven companies, the USB Implementers Forum (USB-IF), with the goal to standardize the connection of peripherals to computers, simplify the configuration of connected devices, permitting faster data transfer rates [13]. USB has since become the standard connector for most computer peripherals, and it is also used in a variety of other devices, including computer peripherals and smartphones. Since the year of its introduction, in 1996, the USB standard has been evolving to support faster transmission speeds and different connector types.

The USB specification is a collection of documents containing all the information about the USB protocol. This includes electrical signaling, physical dimensions of the connectors, protocol layer, and other design aspects. The first iteration of the USB protocol supports two configurations, low-speed mode, which is able to achieve a maximum theoretical speed of 1.5 Mbps, and full-speed mode, increasing the throughput to 12 Mbps. The following iteration, USB 2.0, increases the speeds by a factor of 40, introducing a third configuration, namely high-speed mode [7]. The USB standard continues to be developed to this day, progressively increasing throughput speeds. Table 1 presents the current USB versions and their respective maximum theoretical speeds.

Table 1: USB versions and their maximum theoretical speeds.

USB Version	Mode	Abbreviation	Maximum Speed
USB 1.0	Low Speed	LS	1.5 Mbps
USB 1.0	Full Speed	FS	12 Mbps
USB 2.0	High Speed	HS	480 Mbps
USB 3.0	SuperSpeed	SS	5 Gbps
USB 3.1	SuperSpeed+	SS+	10 Gbps
USB 3.2	SuperSpeed+	SS+	20 Gbps
USB 4.0	-	USB4	40 Gbps

2.2 Design and Architecture of the USB

USB uses a tiered-star topology. Similarly to a master-slave design, only one USB device is is responsible for initiating communication on the bus, and it is called USB host. The topology structure starts from the host and continues in a reverse-tree manner, as Figure 1 depicts. All other peripherals connected to the host, for example mouse, printer or flash drive, are simply referred to, in USB terms, as devices [15]. Upstream traffic means data going from the devices to the host, and downstream traffic is data moving from the host to the devices.

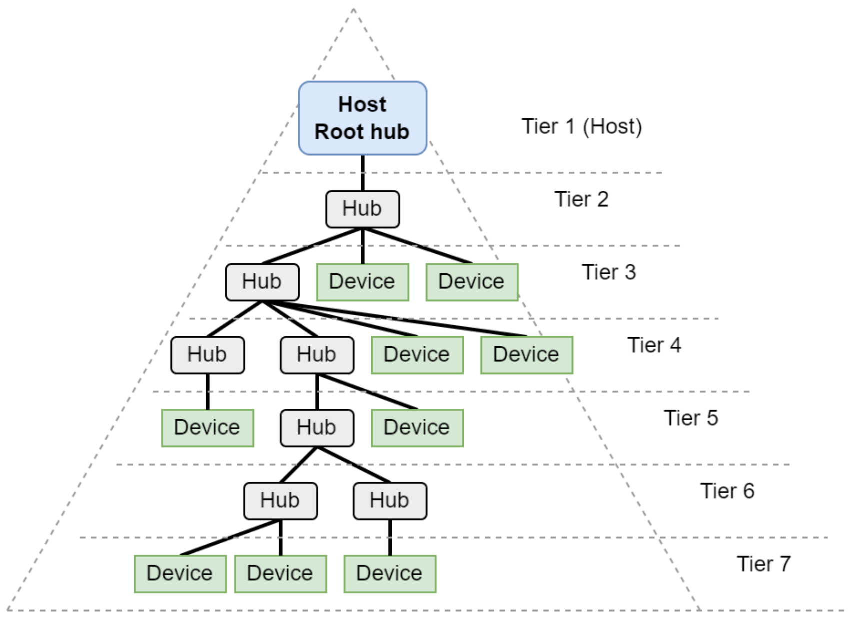


Figure 1: USB tiered-star topology.

A connection between the host and a device is established through a hub or a series of hubs. The host has a root hub attached to it, often integrated to the computer or micro-controller, permitting a direct communication with other devices [8]. Each hub has attachment points called ports, one specifically for upstream connection to another hub, and up to seven downstream ports for connecting other hubs or devices. A maximum of 127 units (hubs and devices) can be part of a single USB topology [15].

2.3 Connector Types

The USB 2.0 specification defines two types of keyed connectors, with the purpose to minimize end user termination problems. Connector type "A" consists of a plug oriented upstream, towards the host, and a receptacle with downstream output from the host or hub. Figures 2 and 3 illustrate a type "A" plug and receptacle, respectively. Connector type "B" is designed to allow devices to have a replaceable cable, consequently "B" plugs are oriented downstream, towards the device, and the receptacles are upstream input to the device or hub. Figures 4 and 5 depict a USB type "B" plug and receptacle, respectively.

With the introduction of USB 3.0, connectors type "A" and "B" received a revision including 5 extra pins to provide higher transfer speeds. A new connector type "C", with 24 pins, was also introduced with the intention of supporting future implementations of the USB protocol. A series of other connectors were created as variations of types "A" and "B" as the USB specification evolved, including "Mini" and "Micro" variants. For the standard versions USB 3.2 and newer, only USB type "C" is available.

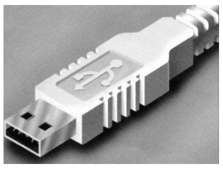


Figure 2: USB type "A" plug.

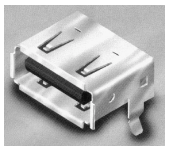


Figure 3: USB type "A" receptacle.

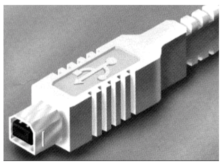


Figure 4: USB type "B" plug.

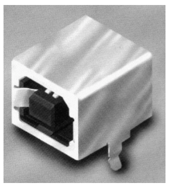


Figure 5: USB type "B" receptacle.

2.4 Physical Interface Wiring

USB 2.0 cables consist of two wires for providing power to the attached device, V_{BUS} at 5 Volts, and GND for ground, as well as two differential wires to provide data [7], as Figure 6 illustrates.

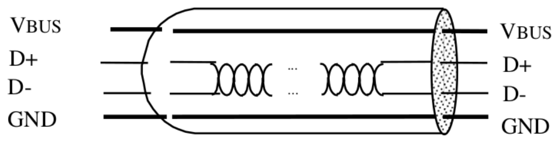


Figure 6: USB 2.0 cable wiring and signals.

The Charger being studied on this thesis interfaces with the monitoring Hub using a proprietary connector that includes the USB differential lines. The connector consists of five pins: a 12 Volts V_{BUS}, D_+ and D_- USB differential lines, a boot mode selection signal, and ground. Figure 7 depicts the Charger connections of the USB interface with the Hub.

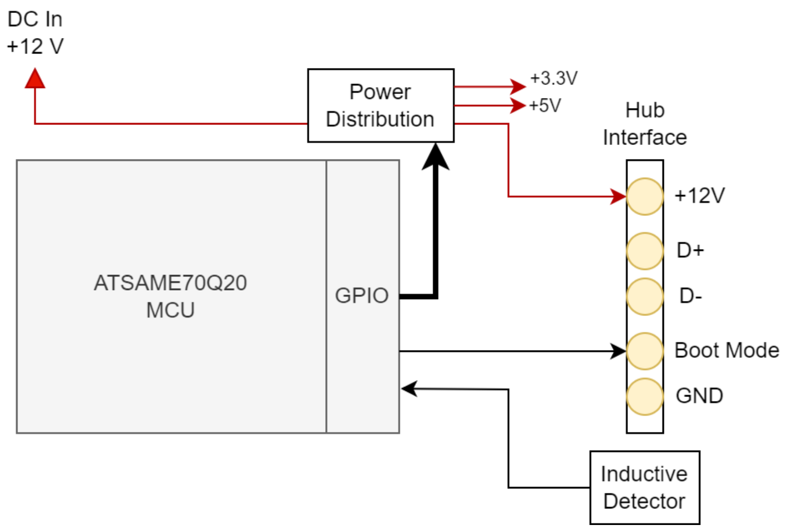


Figure 7: USB interface between Charger and Hub.

2.5 Data Flow

In a USB bus, when the host transmits to a device, every other device on the the bus will receive that data, but only the destination device is able to accept the packets. On the contrary, when the host initiates a request for reading data, only the device which is being requested is allowed reply to the host. There can be no device-to-device communication or multiple devices sending data concurrently to the host [16].

Aside from being responsible for transmitting data, the lines D+ and D- are used for indicating several bus states and device conditions. Signalled conditions include detached, attached, and idle. Indications for reset, end-of-packet, suspend, resume and keep-alive signal are also possible by using specific combinations of D+ and D- states and how long they are held for.

All USB communication events are accomplished from transactions initiated by the host. These transactions occur during a frame, which have a well defined periodic value based on the USB speed of the bus. For example, on full-speed mode the frame is one millisecond, but on high-speed mode the frame is 125 microseconds. Each frame is initiated by a signal indicating the start of frame, or SOF. In order obtain a USB certification of compliance, the host must precisely issue SOFs. Figure 8 illustrates an example of a USB frame and its transactions.

The process of transferring data happens when the host reads or writes data to predefined memory locations on each device called endpoints. A device can have multiple endpoints, and each one has a device-determined direction of data flow. For example, input endpoint one, output endpoint one, input endpoint two, and output endpoint two. Input as endpoint direction always refers to data going into the host, and output, out of the host. For the USB host, endpoints are called pipes.

During a frame, multiple transactions can occur, including three types of packets: token, data, and handshake. Each transaction starts with a token packet indicating

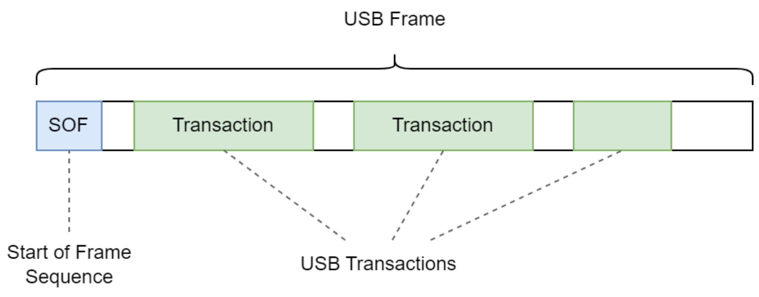


Figure 8: USB frame and transactions.

the type of transaction. There are four kinds of token packets: SOF, which is the Start-of-Frame packet and it is always the first packet of the frame; SETUP, signalling the beginning of a control command towards a device; OUT, for sending payload data to a device; and IN, for receiving payload data from a device. Payload is transferred via data packet, and at the end of each transaction, there can be a handshake packet indicating acknowledged, not-acknowledged or stalled transaction [17].

The USB standard defines three different transfer modes in which the host can communicate with the devices on the bus. *Interrupt transfer* is a highly reliable mode, scheduled on fixed intervals but with limited bandwidth. This transfer is typically used on mice, keyboards and medical devices. *Isochronous transfers* are optimal for high bandwidth applications, where reliability is not a key factor, for example for audio or video streaming. *Bulk transfers* can offer high reliability and bandwidth, given there is enough available capacity on the frame [17], and can be found on mass storage devices and printers.

Bulk and interrupt transactions follow a similar control flow. Figure 9 shows the control flow and error cases for the bulk and interrupt transfers. The host initiates an IN transaction by sending the token to the device and waiting for a reply. If able to, the device replies with data and the host acknowledges it, otherwise it will send a NAK (not-acknowledged) handshake packet indicating it cannot send data, or a STALL handshake packet to inform the endpoint is halted. The OUT transaction is initiated by the host after the OUT token is sent to the device. As the host contains the data to be transfer ed, it can send the data packet immediately followed by the OUT token, and wait for the device's handshake to complete the transaction. In addition to the three handshake packets ACK, NAK, and STALL, the device can reply with a NYET (not yet) packet during bulk output transfers to indicate the transaction is not yet completed [10].

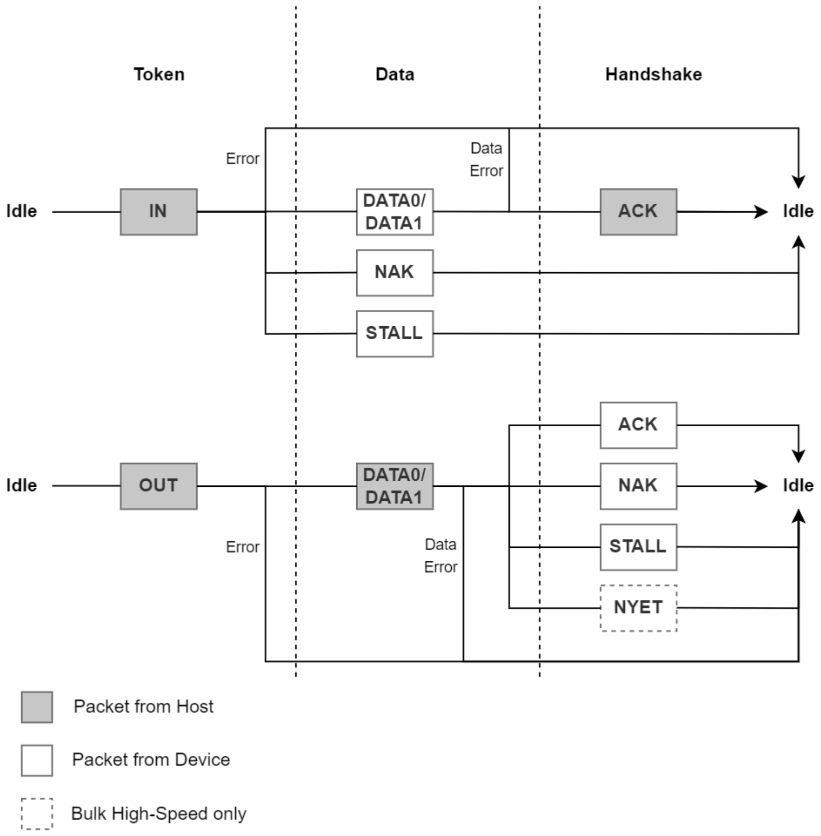


Figure 9: Flow control for bulk and interrupt transactions.

2.6 Device control

Based on the USB protocol, devices are required to provide information about their characteristics and capabilities when they are connected to the bus. This information is exchanged with the USB host during the device's enumeration process. Device information is conveyed by the use of descriptors, containing the device class, subclass, and protocol; vendor and product IDs; the device's power requirements; and the required endpoint configuration. They may also include human-readable strings providing information such as the device's name and manufacturer [18]. During enumeration, the host uses the descriptors information to load the appropriate device drivers and initiate communication. The USB descriptors are defined by USB-IF, providing standard descriptor types and formats that all USB devices must support, as well as some optional descriptors for specific device classes [19].

2.6.1 Device Descriptor

The device descriptor is the first to be accessed by the host and provides basic information about the USB device, such as supported USB versions, device class, vendor and product IDs, as well as the number of configurations the device can assume. The elements of the descriptor can be seen in Table 2.

Table 2: USB device descriptor elements of a C struct, their data types and description.

Data type	Name	Description
uint8_t	bLength	Size of this descriptor in bytes.
uint8_t	bDescriptorType	Descriptor type.
uint16_t	bcdUSB	USB specification release number in BCD format.
uint8_t	bDeviceClass	Device class code.
uint8_t	bDeviceSubClass	Device subclass code.
uint8_t	bDeviceProtocol	Device protocol code.
uint8_t	bMaxPacketSize0	Maximum packet size of endpoint 0 (in bytes).
uint16_t	idVendor	Vendor ID.
uint16_t	idProduct	Product ID.
uint16_t	bcdDevice	Device release number in BCD format.
uint8_t	iManufacturer	Index of the manufacturer string descriptor.
uint8_t	iProduct	Index of the product string descriptor.
uint8_t	iSerialNumber	Index of the serial number string descriptor.
uint8_t	bNumConfigurations	Number of possible configurations for the device.

2.6.2 Configuration Descriptor

The configuration descriptor provides the number of interfaces used and how much power the configuration would required from the host. One single device might have more than one possible configuration, for example, a USB audio interface that supports multiple audio formats may use a different descriptor for each format. The elements of a configuration descriptor are detailed in Table 3.

Table 3: USB configuration descriptor elements of a C struct, their data types and description.

Data Type	Name	Description
uint8_t	bLength	Size of the descriptor in bytes.
uint8_t	bDescriptorType	Descriptor type.
uint16_t	wTotalLength	Length of all descriptors returned along with this configuration descriptor.
uint8_t	bNumInterfaces	Number of interfaces in this configuration.
uint8_t	bConfigurationValue	Value for selecting this configuration.
uint8_t	iConfiguration	Index of the configuration string descriptor.
uint8_t	bmAttributes	Configuration characteristics.
uint8_t	bMaxPower	Maximum power consumption of the device when in this configuration.

2.6.3 Interface Descriptor

The interface descriptor details the interface as a collection of endpoints that define a feature or function of the device. The key elements of this descriptor are the number of endpoints and the USB device class they implement. For example, a USB mouse requires a minimum of one endpoint (IN) for the Human Interface Device (HID) class, whereas a USB flash drive needs two endpoints (IN and OUT) for its Mass Storage device class operation. All elements of the interface descriptor are described in Table 4. The elements *bInterfaceClass* and *bInterfaceSubClass* are codes defined by the USB-IF for each specific device and their intended usage, for example, communication devices (CDC) have the value 0x02 for *bInterfaceClass* and human interface devices (HID) have the value 0x03.

Table 4: USB interface descriptor elements of a C struct, their data types and description.

Data Type	Name	Description
uint8_t	bLength	Size of the descriptor in bytes.
uint8_t	bDescriptorType	Descriptor type.
uint8_t	bInterfaceNumber	Number of the interface in its configuration.
uint8_t	bAlternateSetting	Value to select this alternate interface setting.
uint8_t	bNumEndpoints	Number of endpoints used by the interface (excluding endpoint 0).
uint8_t	bInterfaceClass	Interface class code.
uint8_t	bInterfaceSubClass	Interface subclass code.
uint8_t	bInterfaceProtocol	Interface protocol code.
uint8_t	iInterface	Index of the interface string descriptor.

2.6.4 Endpoint Descriptor

Each endpoint is described by its endpoint address, transfer type (bulk, interrupt, isochronous), and maximum packet size that can be transferred through it. The elements of an endpoint descriptor are shown on Table 5. The most significant bit of *bEndpointAddress* indicates the direction of the endpoint (0: OUT, 1: IN), and the first four least significant bits contain the endpoint number. The element *bmAttributes* contain information about the transfer type of that endpoint on the first two least significant bits (00: control, 01: isochronous, 10: bulk, and 11: interrupt) [9].

Table 5: USB Endpoint Descriptor elements of a C struct, their data types and description.

Data Type	Name	Description
uint8_t	bLength	Size of the descriptor in bytes.
uint8_t	bDescriptorType	Descriptor type.
uint8_t	bEndpointAddress	Address and direction of the endpoint.
uint8_t	bmAttributes	Endpoint type and additional characteristics (for isochronous endpoints).
uint16_t	wMaxPacketSize	Maximum packet size (in bytes) of the endpoint.
uint8_t	bInterval	Polling rate of the endpoint.

The USB enumeration is the process in which a USB device is identified, configured and initialized by the USB host controller. This process allows the host to load the correct drivers for each specific USB device. The enumeration involves a series of steps that uses the electric signals on the USB cables as well as the information obtained by the device descriptors exchange [11]. Figure 10 shows the typical enumeration steps after a USB device is connected.

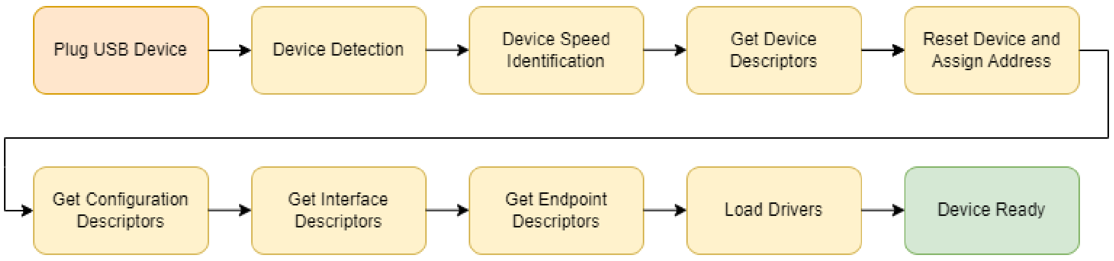

Plug USB Device
Device Detection
Device Speed Identification
Get Device Descriptors
Reset Device and Assign Address
Get Configuration Descriptors
Get Interface Descriptors
Get Endpoint Descriptors
Load Drivers
Device Ready

Figure 10: Enumeration steps of a USB device performed by the USB host.

2.7 USB Emulation Models

The USB standard characterizes three classes to define communication devices: Communication Device Class (CDC), Communication Interface Class and Data Interface Class. This architectural definition was created to support any communication device to use the USB protocol as a mean of data transfer [5]. For telecommunication devices and networking services it is possible to implement the CDC models to use conventional TCP/IP networking over USB.

When a device is connected to the USB bus, the host will initiate the enumeration process in order to configure the device and enable application data exchange. During this negotiation, the device provides the host the type of communication it requires via the interface descriptors. The CDC interface allows the emulation of typical communication and networking protocols via the USB bus. For example, a device can request a UART link to the host via a virtual COM-port using the Abstract Control Model (CDC-ACM), or an Ethernet adapter can exchange IEEE 802.3 packets with the host using the Network Control Model (CDC-NCM). Other CDC models are Ethernet Emulation Model (CDC-EEM) and Ethernet Control Model (CDC-ECM). Microsoft has its own protocol for Ethernet communication over USB called Remote Network Driver Interface Specification (RNDIS). Each model uses different methods to encapsulate the communication packets on USB transactions.

A Communication Interface Class of type Abstract Control Model, or CDC-ACM, consists of a minimum of two endpoints; one for managing the elements of serial communication and the other to implement notifications. Two extra endpoints can be used to implement the data transfer capabilities, functioning as input and output channels [20]. An application can implement a UART driver using the CDC-ACM model to emulate this type of communication over the USB bus.

In contrast to a telecommunication or UART device, a networking device will always have at least one Data Class Interface associated to it, assigning one input and one output endpoint for exchanging network traffic. One extra alternative endpoint must be configured to initiate any transactions. The Communication Class Interface may be utilized for providing device configuration and statistics collection for the device, but it is not mandatory to be active. The CDC-ECM subclass enables the host to send or receive one Ethernet packet per USB packet. A USB packet in high-speed mode can have maximum size of 512 bytes, but a typical Ethernet frame can be as larger as 1514 bytes. The CDC-ECM protocol enables the negotiation of the maximum segment size, allowing one large Ethernet frame to be transfer over multiple USB transactions, with a following zero-length packet to delimit the end of the frame [21].

The Ethernet Emulation Model is a simpler subclass compared to the CDC-ECM, as it only requires a pair of endpoint for sending and receiving frames. This model has the capability of encapsulating multiple Ethernet packets into a single USB transaction, potentially improving the throughput over ECM implementations [22]. CDC-EEM packets contain a header to determine the length of the included Ethernet frames and a checksum. This extra overhead should be taken into consideration when using this model as it could increase the necessary processing of the frames.

In summary, Chapter 2 introduced the USB standard, providing an overview of its design, architecture, and data flow. It discussed connector types, physical interface wiring, as well as how the device descriptors can be used to detect and enumerate a USB device. The chapter also described the Ethernet over USB emulation models that are relevant to this thesis.

3 Case Study

This chapter describes the current platform under study and how the system as a whole is designed to operate, as well as introduces the motivations for the implementation of the USB to Ethernet interface. The development plan and thesis methodology are also discussed in the following sections.

3.1 Current Platform

In 2022, GE HealthCare introduced to the market a new patient monitoring product called Portrait Mobile. The initial release consists of two wearable mobile sensors for acquiring respiration rate, pulse rate, and oxygen saturation levels. The sensors are powered by removable batteries that can be attached to a charging station. The batteries also act as a radio to transmit and receive control and monitoring data. The **Hub** is a mobile monitoring viewer that connects to the sensors and manages the patient's vitals. The Hub may also be docked to the **Charger** to refill its battery. Figure 11 presents the complete Portrait Mobile system.

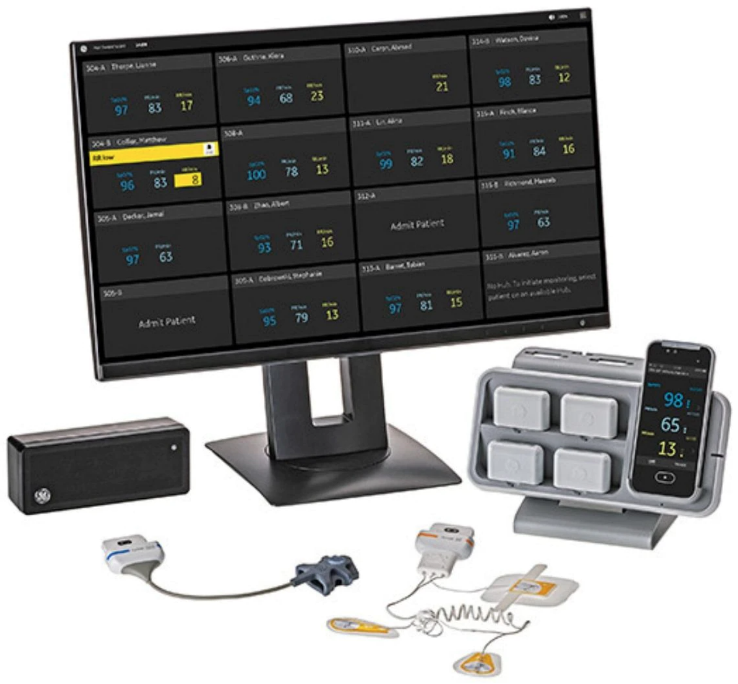


Figure 11: GE HealthCare Portrait Mobile monitoring system, including the Viewer, alarm box, SpO2 and respiration sensors, Charger, Sensor Batteries, and Hub.

As a patient is admitted, their monitoring is initiated by creating a profile on the Hub and pairing the sensors to it. From this point, the Hub will communicate through Wi-Fi with the core service **Axone**, which can run locally on a server at the hospital, or as a cloud service. There are two ways to visualise and monitor the patient's sensors. From the Hub, typically located on the bedside or close to the patient, medical staff can have quick access to vitals values, trends and alarm settings. Alternatively, multiple patients can be monitored simultaneously from a centralized **Viewer**, receiving data from the Axone service to provide different views of the data, trends and alarm configurations.

The use of wireless technology to transmit monitoring data can bring multiple benefits to patients and medical staff, such as increased mobility when moving in or out of the bedside, and allowing clinicians to remotely follow the patient health. Although the communication between the sensors and the Hub uses a robust proprietary protocol, the Hub shares the airway with all other Wi-Fi enabled devices on the premises when communicating with the Axone service. Given the nature of the Wi-Fi protocol, it is possible to experience data loss or disruption depending on the infrastructure and how busy the area may be.

A wired LAN connection from the Hub to the network would mitigate problems related to poor wireless connectivity, giving clinicians and patients more confidence that the monitoring data will be reliably transmitted. This thesis proposes the solution of LAN connectivity and proves it can be a reliable way to convey clinical data.

3.1.1 Sensor Data

The initial version of Portrait Mobile can be used to monitor three types of clinical data: blood oxygenation, pulse and respiration rates. The sensors are capable of collecting, processing and filtering the data before sending to the Hub. This is an efficient data collection process that optimizes the total size of clinical data to be transmitted. In a standard operating condition, the SpO2 Sensor is expected to transmit 4 KB of data per second, and the Respiration Sensor to transmit 2 KB of data per second. In addition to the sensors data, the Hub calculates trends which are sent to the network at longer intervals, averaging 4 KB per second.

3.1.2 Back-filling Offline Data

If the Hub disconnects from the wireless network, it is able to store up to two hours of sensor data and trends information. When connection is restored, all the data stored is back-filled to the Axone service, to avoid having gaps on monitoring data. Given the nominal data rates, there can be a maximum of 72 MB of data to be transferred in a retrospective way. Under optimal wireless network conditions, the back-filling transfer would take only a few seconds, but in cases of poor wireless signal or heavy interference, the transfer could take minutes to complete, potentially causing even further disruption on the monitoring service.

Although the service disruption while back-filling data would be undesirable, a more significant issue is relying entirely on a wireless communication for transmitting vital signals from patients. As new sensor types are added to the Portrait Mobile system, more data would need to be transmitted, for example, some of the future iterations of the product may include sensors producing considerably more data than the current ones. Increasing wireless data connectivity may result in more data interruptions and failures to trigger alarms during critical conditions.

The current design of the product is able to provide sufficient connectivity capability via Wireless Local Area Network for the initial applications and sensor types. As the product develops and more sensors and extra features are introduced, a redundant wired Local Area Network connection would contribute to a more stable and reliable connection.

3.2 Development Plan

The Portrait Mobile Charger uses the microcontroller ATSAME70Q20, originally developed by semiconductor manufacturing company Atmel Corporation, which now belongs to Microchip Technology Inc. The microcontroller is based on the 32-bit ARM Cortex-M7 processor and includes multiple peripherals such as 10/100Mbps Ethernet MAC, high-speed USB host and device, and UART [24].

The Charger software is written in C programming language (ISO/IEC 9899:2018) and it is composed of multiple independent tasks, controlling different functionalities. The software execution is managed by the real-time operating system FreeRTOS, a kernel specifically designed for embedded devices and micro-controllers, freely distributed under the MIT License and currently maintained by Amazon Web Services (AWS) [25]. Because the software is expected to run for long periods of time, to prevent memory leaks during runtime, most of the resources are allocated during build time or at system start-up.

The current version of the Charger software includes support for USB emulation of UART traffic by using the CDC-ACM protocol. This is how the Hub is able to communicate with the Charger to perform tasks including charging monitoring and software updates. The addition of an Ethernet to USB bridge will allow the Hub to use the Ethernet port of the charger as an alternative network interface to the Wi-Fi connectivity.

This section will propose a development plan to outline the objectives, requirements, and methodology for the implementation of the USB to Ethernet bridge on the Charger.

3.2.1 Objectives

The objective of this thesis is to design and implement a software feature for the Charger to enable a communication link between the USB and Ethernet ports of the micro-controller, ultimately allowing the Hub to use the Charger's Ethernet port as its own network interface. Furthermore, an evaluation will be provided on the performance of the implemented solution in terms of data transfer rates, latency, and stability under different circumstances.

By meeting these objectives, this thesis will contribute to the development of a solution for providing the Hub a more robust connection to the monitoring services and allowing the expansion of the Portrait Mobile capabilities. Moreover, this will contribute with insights into the use of micro-controllers for similar applications.

3.2.2 Requirements

As a medical device class II, the Portrait Mobile Charger must comply with regulatory specifications suggested by the FDA and EMA, in order to make the product available to the market. For example, in the United States, some of the regulatory requirements are the premarket notification 510(k), labeling requirements, and quality system (QS) regulations [6]. The software work done in this thesis affects primarily QS requirements, as any design changes must be documented and their impact re-analyzed. This is typically done with thorough testing supported by a Failure Mode and Effect Analysis (FMEA), leading to defects prevention and enhancements on safety [26]. Due to the high complexity and large scale of the aforementioned topics, the scope of the requirements for this thesis will be limited to achieving the desired functionality and performance, which will be evaluated by a set of test scenarios.

The functional requirements for this project are: to provide the Charger the capability of being a USB host, enabling the Hub to be enumerated as a USB device; to convert USB packets from the Hub into Ethernet frames (and vice-versa); to implement the Ethernet protocols TCP/IP and UDP; and to support USB high-speed standard.

Performance and stability metrics are part of the the non-functional requirements, which expect the Charger to operate with low latency data transfers, high stability and reliability. Appropriate documentation on the test procedures and results is needed to validate the non-functional requirements.

3.2.3 Hardware

In order to implement a bridge between USB and Ethernet using the ATSAME70Q20 microcontroller, a deep understanding on its components and how a software is able to interact with such peripherals is needed. The USB functionality is provided by the USB High-Speed Interface (USBHS), and the Ethernet Media Access Controller (GMAC) peripheral allows the implementation of IEEE 802.3 networking. The software to be designed on this thesis must be able to provide capability of interacting with these two peripherals so data could be moved from the USB device to the GMAC peripheral in form of Ethernet packets, and vice-versa.

3.2.3.1 USBHS - USB High-Speed Interface: USBHS is the USB peripheral on the ATSAME70Q20 microchip, complying with the Universal Serial Bus 2.0 specification. The peripheral supports high-speed, full-speed and Low-Speed communication. It has 9 pipes/endpoints with 4096 bytes of embedded dual-port RAM (DPRAM) available for managing traffic data. Additionally, the device includes dedicated Direct Memory Access (DMA) channels for fast and efficient data transfer [24].

Enabling Ethernet emulation using the USB peripheral requires the USB stack running on the MCU to be configured for supporting either the CDC-ECM or CDC-EEM protocols. In this application, the Charger will be configured as a USB host. When a device is connected to the USB port of the ATSAME70Q20, the enumeration process will initiate and a series of descriptors will be exchanged so the USB host can correctly configure the connected device. After enumeration, the necessary pipes on the USBHS peripheral are allocated and any subsequent USB data sent from the device is available to be captured by the USB driver, or data can be queued into the output pipe to be sent to the device.

3.2.3.2 GMAC - Ethernet Media Access Controller: The GMAC peripheral allows the microcontroller to implement a 10/100 Mbps Ethernet MAC connection, compatible with the IEEE 802.3 standard. Other key characteristics of the GMAC include full and half duplex operation, DMA interface to external memory, interrupt generation on receive and transmit completion, as well as 8 KB transmit and 4 KB receive RAM.

The driver controlling the GMAC allocates two buffer lists, one for receiving and one for transmitting. The lists are tied to the DMA configuration registers for fast data transmission between the FIFO and internal buffers. When the GMAC driver is initialized, a task controlling Ethernet data traffic can be implemented to read queued data from the Ethernet port, or queue data to be sent by the peripheral.

3.2.4 Software

The practical work on the software consists of two parts: integrate the software stack for the GMAC and USBHS peripherals, making the necessary adjustments to fit the application of the Ethernet-USB data link; and implement a software task to support data exchange between the peripherals at the rated transfer speeds. The current functionality of the Charger, including the UART emulation via USB, must remain as is. The software must have a clear design and concise architecture, which will be presented on the following chapter.

The methodology of this thesis consists of developing a software solution for the problem in question: how to provide an alternative network interface for the Portrait Mobile Hub via the Charger? The solution will be analyzed by performing a selection of stability and performance tests, to conclude if the final product meets the requirements. More details on how the problem will be solved and testing methodology is presented on the following section.

3.3 Thesis Methodology

Fundamentally, the purpose of this thesis is to use the MCU ATSAME70Q20 on the Charger as a USB to Ethernet converter. Devices with this functionality have been widely available for the consumer market, and even as integrated circuit packets, like Microchip's LAN9500A, for direct use on PCB assemblies. The use of off-the-shelf components to enable Ethernet communication when the Hub is docked to the Charger is a relatively easy solution to implement, requiring a PCB redesign and minor software changes. However, the introduction of a new IC could greatly increase the production costs on the long term. At the time this was written, the cost of a LAN9500A IC was approximately $5,11 [27], which could lead to millions of dollars of extra cost during the expected lifetime of the product. Another incentive for implementing the Ethernet bridge on the Charger MCU is that older hardware designs could still be used, only requiring a software update to enable the new feature.

Atmel provides a software package to be used for programming its microcontrollers. This package contains a common USB stack which is taken into use for the implementation of the USB host functionality on the Charger. The USB host stack consists of a Hardware Abstraction Layer (HAL), a device driver and an Application Programming Interface (API). The HAL interacts directly with the USBHS peripheral, for example providing functions to configure, start, suspend, and access register data. The device driver provides high level commands for specific devices, and utilizes the HAL to provide hardware instructions to be used by the USBHS peripheral [28]. The API is a collection of functions used to interface with the USB host at a higher layer of abstraction.

The Charger should be able to switch between USB host and USB device depending on the use case. The Charger has a USB port on its back used for servicing and debugging, as Figure 12 shows. When the USB port of the Charger is connected to a PC, the MCU should behave as a USB device. In this case, it is possible to perform software updates on the Charger or on the devices connected to it (Sensor Batteries and Hub). When there is no PC connection, the Charger should behave as a USB host, enumerating the Hub as a device on the bus, and providing the LAN connectivity via CDC-ECM/CDC-EEM protocol.

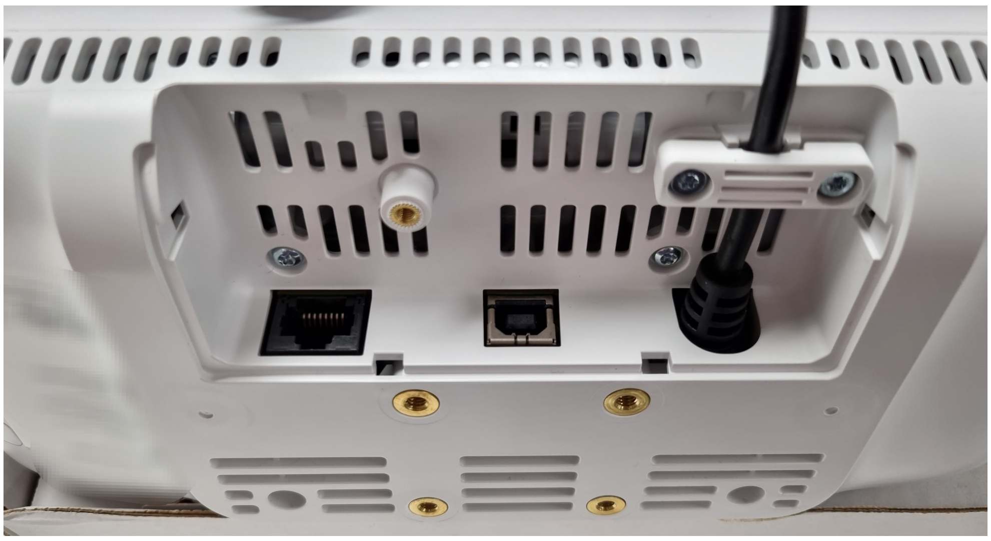


Figure 12: Connectors at the back of the Portrait Mobile Charger. Left to right: Ethernet, USB service port, and DC input.

The Hub uses the CDC-ACM protocol to communicate with the Charger through the 5-pin slot interface. The current software identifies when the Hub is placed on the charging slot and enumerates it as a UART-capable communication device. One of the modifications needed for the Atmel USB software stack is to add support for CDC-ECM and CDC-EEM enumeration. In addition, further optimization is needed to be able to support the concurrent operation of CDC-ACM and CDC-ECM/CDC-EEM.

The USB endpoint and pipe configuration should be carefully implemented in order to avoid slow downs due to traffic congestion or excessive interrupts.

It is good practice for a software development project to have a clear idea of what the architecture and general software design. The areas of the software concerning USB and Ethernet connectivity must be clearly identified, providing information about their purpose and how they interact with each other as well as with other tasks. The way data flows from one peripheral, into the internal buffers, then continue toward the other peripherals will be described in the software design section of the following chapter.

Achieving transfer speeds close to the rated bit-rate of the peripherals will require effective data handling and optimizations at driver and application level. The software design should include methods for managing data traffic between CDC-ACM and CDC-ECM/CDC-EEM protocols, avoid excessive memory copying and taking advantage of DMA transfers to reduce latency.

Once the software is implemented and the data bridge between USB and Ethernet is enabled for the Hub, a selection of tests will be executed to determine how well the performance of the Ethernet link is. The most valuable metrics for this application are the bandwidth, latency and link stability. Transfers speeds should be close to 100 Mbps, which is the bottleneck on the data path caused by the GMAC peripheral, rated for 10/100 Mbps. The bandwidth will be measured with **iPerf3**, which is an open-source tool used for determining the maximum achievable bandwidth on IP networks. iPerf3 supports configurable data packets as well as internet protocols TCP and UDP, providing a diversity of test scenarios. For latency, a simple tool called **Ping** can be used to measure how long it takes for a message to travel from the Hub to the local network and back. Although simple to use, Ping offers several options for customizing the tests, for example performing flooding operations. Other tools developed by GE Healthcare will be used to test system stability, how the Charger performs under network stress, and UART communication with the Hub via CDC-ACM.

The solution for the Ethernet bridge will be evaluated by the aforementioned tests to indicate if the new feature performs adequately enough to be part of the Portrait Mobile Charger device. The tests are designed to cover all the required functionality of the LAN connectivity for the Charger, including real use cases and stress testing.

This concludes the description of the case study of this thesis. The Charger device was introduced, including how it interacts with the other components of the Portrait Mobile system. The development plan to solve the research question was presented, where the objectives and requirements were proposed, in order to implement and test the USB to Ethernet software bridge using the Charger's microcontroller.

4 Solution Development

The implementation of the data bridge can be divided into three areas of development: the USB task, which controls the USB-related operations and the drivers for the communication class devices; the Ethernet task, responsible for interfacing with the GMAC peripheral and providing the translation layer between USB packets and Ethernet frames; and the data flow management, to ensure the data transfer between peripherals is done in orderly and efficient way. These areas will be further discussed in this chapter, showing how the data bridge was designed and implemented.

With the MCU acting as a bridge between USB and Ethernet, the data packets going through it undergo a transformation in order to be recognized and accepted by their respective peripherals. When a packet is received by the USBHS peripheral on its USB interface, it gets stored on static random access memory (SRAM) until is ready to be used by the Ethernet driver. Depending on the Ethernet emulation type, the USB packet contains different types of header, so it must be parsed before sending to the GMAC peripheral. Moreover, when an Ethernet packet arrives at the GMAC interface, it gets encapsulated by an appropriate CDC header and stored in SRAM until the USB driver is ready to transmit it. The transmission interface between USB and Ethernet and the transformations the packets undergo are described on the architectural diagram in Figure 13.

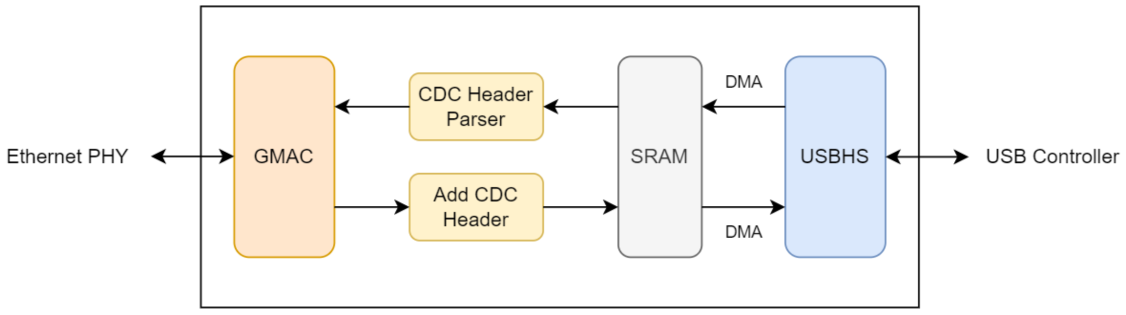


Figure 13: Architecture diagram of the Ethernet to USB data path bridge.

The Charger software is operated by the real-time scheduler FreeRTOS. This is kernel of the system and it allows the application to have multiple independent tasks running as separate threads. Examples of different tasks are charging management, LED indications, temperature monitoring, and USB communication. It is a responsibility of the scheduler to prioritize and execute these tasks based on their priority. On a single-core MCU, such as the ATSAME70Q20, only one thread can be executed at a time. The scheduler will maintain a list of active threads ordered by their priority and give them CPU time accordingly. This ensures time-critical hard real-time operations are always executed before soft real-time tasks, being one of the fundamental functionalities of a real-time embedded system [30]. During system startup, the hardware components are initialized and each task is configured with a main function to run as well as its priority. There are over 30 tasks on the Charger software, but in this thesis the main focus will be on the tasks that govern Ethernet and USB operations.

4.1 USB Task

The USB task starts by initializing the Atmel USB stack, then continues to run in a loop performing monitoring of USB-related events. To follow the requirements of USB hierarchy, where only one host is allowed per bus, a USB multiplexer controls the connections of the devices. When the Charger is powered on and a Hub is docked to it, the multiplexer will enable the connection between the Hub and the MCU, starting the USB host on the Charger. This scenario is illustrated by Figure 15. If a PC is connected to the USB service port, the Charger can no longer be a host, so the multiplexer selects the Hub line and reconnects the MCU via the FTDI chip, as shown in Figure 14. This stops the USB host on the Charger, which becomes a USB device under the PC connection tree. On the last case, when the Charger is connected to the PC but there is no Hub docked, the MCU will be routed to the PC, but acting as a USB device for FTDI UART communication. This configuration is presented in Figure 16.

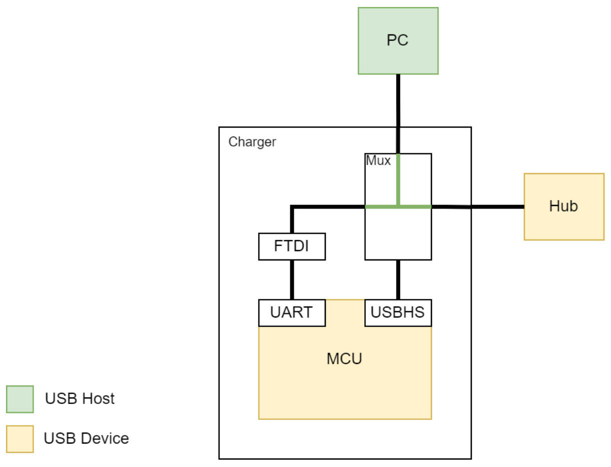


Figure 14: Multiplexer configuration when Hub is docked and Charger is connected to PC via USB cable. PC is the USB host.

The Hub should only have access to the LAN when there is no PC connected to the Charger. The event of connecting the Charger to the PC can be detected by the voltage on the USB line V_{BUS}, triggering an interrupt whenever the voltage goes past the threshold for logic level "high". The Hub detection is not done by reading the V_{BUS} voltage on the Hub connector, as the Hub is a USB device and can only use the V_{BUS} line as a sink [12]. The detection is done by a inductive sensor located under the Hub slot, which can be queried by software to indicate the presence of the Hub. With these two detection mechanisms the USB task is capable of selecting the multiplexer channels accordingly and operating the USB host stack on the MCU.

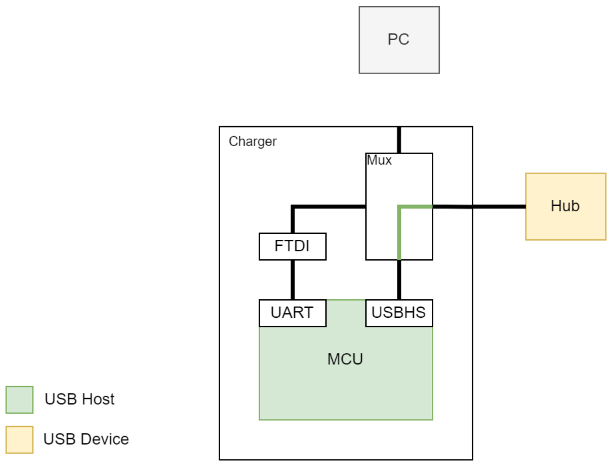


Figure 15: Multiplexer configuration when Hub is docked and Charger is disconnected from PC. MCU is the USB host.

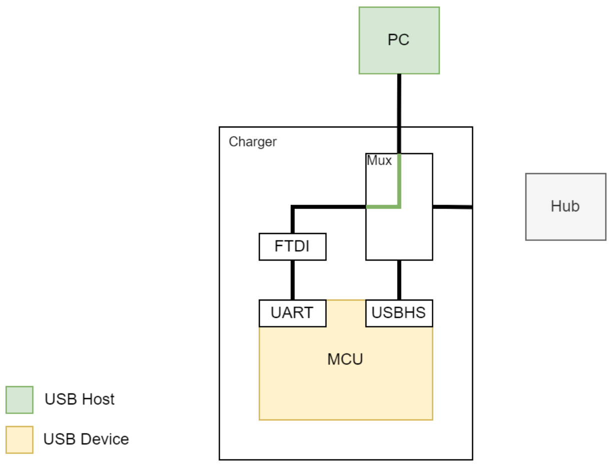


Figure 16: Multiplexer configuration when Hub is not docked and Charger is connected to PC via USB cable. PC is the USB host.

4.1.1 USB Host Stack

The Atmel USB stack for the ATSAME70Q20 micro-controller provides a simple API for the USB task with the functions *USBH_start()* and *USBH_stop()*. Whenever the Hub is connected to the MCU via USB multiplexer, the function *USBH_start()* is called to configure the MCU as a USB host. If a device is detected on the USB bus, the MCU will initiate a series of descriptor exchanges and attempt to configure the new device. If the USB multiplexer disconnects the MCU and the Hub, *USBH_stop()* is executed, terminating the MCU as USB host and cleaning all device descriptors information.

4.1.1.1 CDC-ACM on the Base Charger Design: The Charger application uses the UART communication interface to exchange messages with the devices connected to it, i.e. PC, Sensor Batteries and Hub. The messages follow a common protocol used across all Portrait Mobile devices. When the Hub is docked to the Charger and the MCU is in USB host mode, the Hub gets enumerated as a CDC-ACM USB device, allowing the Charger's application to communicate with it via UART messages. During the enumeration process, the USBHS allocates three pipes for the Hub: a bulk IN and a bulk OUT pipes for exchanging data with the host, as well as an interrupt IN pipe the host uses to receive notifications from the device. The Atmel USB stack allocates pipes to the corresponding device's endpoints in ascending order as Endpoint Descriptors are processed. There is no specific order for setting the priority of the pipes, and it is the device's responsibility to provide the Endpoint Descriptors in the correct order. The following logs are from the moment the Hub is docked to the Charger and the enumeration process begins.

```
>USB EVENT
>USB enum: reset USB line
>USB enum: idle 20ms
>USB enum: reset USB line
>USB enum: idle 100ms
>USB enum: get device descriptor
>USB enum: setting address
>USB enum: get configuration descriptor
>USB enum: getting full configuration descriptor
>USB enum: setting configuration
>USB enum: get device descriptor
>USB enum: setting address
>USB enum: get configuration descriptor
>USB enum: getting full configuration descriptor
>USB enum: setting configuration
>ACM Found
>ItfDesc. Class: 2, SubClass: 2, Protocol: 1, Num endpoints: 1
>EpDesc. Ep Address: 130, Attributes: 3, Interval: 9
>ep 2 -> pipe 1
>ItfDesc. Class: 10, SubClass: 0, Protocol: 0, Num endpoints: 2
>EpDesc. Ep Address: 129, Attributes: 2, Interval: 0
>ep 1 -> pipe 2
>EpDesc. Ep Address: 1, Attributes: 2, Interval: 0
>ep 1 -> pipe 3
>Device Driver installed
>USB enum: enabling UHIs
>ACM PLUG
>APP LOG 52: USB TASK: USB device was enumerated in hub slot, VID 0x1901, PID 0x1b
```

When placed on the Charger slot, the Hub gets routed to the MCU by the USB multiplexer, triggering an event that initiates the enumeration process. As the host enumerates the newly added device, the Interface and Endpoint Descriptors are identified and the peripheral's pipes are configured. Class number two indicates a communication interface for a CDC device. Subclass two is the USB-IF definition for an Abstraction Control Model interface and it contains one input interrupt endpoint. The communication interface is followed by a data interface (Class 10), and it contains the description of the endpoints used for transferring data between the host and device [5]. From the example, the first endpoint to be assign has the address $130 = 0b10000010$, meaning it is an input endpoint (bit7 = 0b1) with assigned number two (bits3:0 =

0b0010). The attribute value 3 = $0b00000011$ indicates it is an interrupt endpoint. The Hub's IN interrupt endpoint two then gets assigned to pipe number one of the USBHS. Similarly, the two following data bulk endpoints IN (bit7 = 0b1) and OUT (bit7 = 0b0), with address one, get assigned to pipes two and three, respectively. The enumeration is concluded once all endpoints are assigned to their corresponding USBHS pipes and the CDC-ACM driver is loaded.

The USBHS has a total of 10 pipes, with the first one being reserved for USB host control operations. During runtime, on every USB frame (125μs in HS, $1ms$ in FS) a handler is called to perform any pending read/write operations on the pipes. The transfer operations are executed following the pipe number order, i.e. the control pipe zero is handled first and pipe nine, last. This default priority is not an issue if the only communication needed by the device is UART, which has relatively low transfer speeds compared to USB 2.0. However, the addition of a second Communication Device Class to handle IP network exchanges on the same device could cause traffic issues on the USBHS peripheral, requiring careful prioritization to avoid either protocol blocking the other.

4.1.1.2 Integrating the CDC-ECM/CDC-EEM Models: The Hub runs its application on top of a Linux operating system. When part of a USB bus, the Linux kernel on the Hub (Yocto Project [31]) provides the host all the supported functions as Descriptors. The Hub is configured to have two USB functions: CDC-ACM, the Communication Class for using UART, and CDC-ECM (or CDC-EEM, depending on the configuration), a second Communication Class for Ethernet emulation. To enable the full communication functionality of the Hub, both CDCs must be enumerated by the MCU, and a driver for handling the Ethernet emulation must be programmed.

The Atmel USB stack includes a USB Host Controller (UHC), that provides a high-level abstraction of the host. This module can be used to control the state of the host, for example, starting, suspending, or resuming operations. The UHC receives from the application a list of USB interfaces the host is expected to support. This list is part of the USB Host Interface (UHI) and it can include multiple interfaces from different classes, such as CDC-ACM, CDC-ECM, Mass-storage Class (MSC), and Human Interface Device (HID). Each interface on the list must provide their core functionality with callback functions the USBHS will run during the device operation.

The driver to handle Ethernet emulation models was implemented to provide the necessary UHI functions to the controller. The core elements of the driver consists of four functions: *install*, *enable*, *uninstall*, and *sof_notify*.

- ***install***: Instantiates the device being installed, keeping information regarding data interface (bulk transfers), communication interface (interrupt transfers), and endpoint addresses. The USBHS pipe configuration is done during device installation.

- ***enable***: Prepares the device to run, notifies other components about the start of operation, and calls the function *sof_notify* to initialize data transfers.

- ***uninstall***: Deletes the device instance by freeing the memory and resetting counters associated to it. This also frees the USBHS pipes that are no longer in use.

- ***sof_notify***: This function is first called during UHI enable, and periodically after every Start of Frame (SOF) synchronization marker. The SOF is a special packet sent by the USB host to all connected devices once every tick (125μs in HS mode). Read and write operations are done in this context by requesting the USBHS to perform an endpoint-pipe transaction.

The integration of CDC-ECM to the Charger communication design, requires adding the newly implemented driver to the list of interfaces supported by the application. The UHI would now have two items: CDC-ACM and CDC-ECM, and the application can select the appropriate driver to install when enumerating the Hub. In this scenario, the challenge of managing data traffic on the peripheral's pipes emerges, because the USB task would need to process requests for both UART and Ethernet communication while sharing the resources of the USBHS. It was observed that enabling both UHIs and letting the USB stack handle the pipe allocation prioritization, leading to the configuration shown in Figure 17, results in the CDC-ACM input pipes completely blocking the Ethernet communication. In the case where the device does not have data to send, as the host initiates IN transactions on the first CDC-ACM pipe, the device replies with a NAK packet to indicate it does not have any data, and this creates a cycle of the host attempting to read until there is data available. This scenario blocks all the pipes that have priority lower than the first input.

Pipe 0	Pipe 1	Pipe 2	Pipe 3	Pipe 4	Pipe 5	. . .
Control	ACM INT IN	ACM BULK IN	ACM BULK OUT	ECM BULK IN	ECM BULK OUT	

Figure 17: USBHS pipe allocation when the USB stack configures endpoints with default prioritization.

To address this issue, the Atmel USB stack function *USBH_HAL_ConfigurePipe()* was modified to prioritize the execution of OUT pipes. The new USBHS pipe configuration presented in Figure 18, allows the high priority write operations to run in every micro-frame, and lower priority input pipes to attempt to read data if the device is ready.

Pipe 0	Pipe 1	Pipe 2	Pipe 3	Pipe 4	Pipe 5	. . .
Control	ACM BULK OUT	ECM BULK OUT	ECM BULK IN	ACM BULK IN	ACM INT IN	

Figure 18: USBHS pipe allocation with modified USB stack, prioritizing OUT pipes execution.

The following logs show the Hub enumeration process after the CDC-ECM driver was added to the UHI list and pipe prioritization applied to the Hardware Abstraction Layer. The Subclass for Ethernet Control Model is defined by the USB-IF as the value six. Note that even though that interface includes one interrupt input endpoint, it is not configured, as it has no use in this application. Only data interface endpoints (Class 10) are configured.

```
>USB EVENT
>USB enum: reset USB line
...
>USB enum: setting configuration
>ACM Found
>ItfDesc. Class: 2, SubClass: 2, Protocol: 1, Num endpoints: 1
>EpDesc. bEndpointAddress: 130, bmAttributes: 3, bInterval: 9
>ep 2 -> pipe 5
>ItfDesc. Class: 10, SubClass: 0, Protocol: 0, Num endpoints: 2
>EpDesc. bEndpointAddress: 129, bmAttributes: 2, bInterval: 0
>ep 1 -> pipe 4
>EpDesc. bEndpointAddress: 1, bmAttributes: 2, bInterval: 0
>ep 1 -> pipe 1
>Device Driver installed
>ECM Found
>ItfDesc. Class: 2, SubClass: 6, Protocol: 0, Num endpoints: 1
>EpDesc. bEndpointAddress: 132, bmAttributes: 3, bInterval: 9
>ItfDesc. Class: 10, SubClass: 0, Protocol: 0, Num endpoints: 2
>EpDesc. bEndpointAddress: 131, bmAttributes: 2, bInterval: 0
>ep 3 -> pipe 3
>EpDesc. bEndpointAddress: 2, bmAttributes: 2, bInterval: 0
>ep 2 -> pipe 2
>Device Driver installed
>USB enum: enabling UHIs
>ACM PLUG
>ECM PLUG
>APP LOG 52: USB TASK: USB device was enumerated in hub slot, VID 0x1901, PID 0x1b
>APP LOG 52: USB TASK: USB device was enumerated in hub slot, VID 0x1901, PID 0x1b
```

4.2 Ethernet Task

The Ethernet task provides an interface between the GMAC peripheral and the charger application. Ethernet access is done by utilizing functions of the GMAC driver, provided by the Atmel ATSAME70Q20 Ethernet software stack, to initialize and perform read/write operations on the GMAC peripheral. During runtime, as packets arrive through the physical layer interface, they are read, parsed according the the Ethernet emulation configuration, and stored in SRAM for later use by the USB task. On the contrary, if a packet is being sent from the USB device, the Ethernet task parses and queues it to the GMAC for transmission.

As discussed in section 2.7 about USB Emulation Models, depending on the Ethernet emulation type the Charger is configured to handle, the header that encapsulates the IEEE 802.3 frames can have different contents. The only packet manipulation done by the Ethernet task is the parsing of the USB CDC-ECM/CDC-EEM headers when reading from, or writing to the internal SRAM buffers.

Ethernet Emulation Model packets have the capability of encapsulating multiple Ethernet frames. That is achieved by prepending each encapsulated Ethernet frame with an EEM header containing a message type bit, a CRC usage bit, and the length of the frame [22], as depicted in Figure 19. The bit *bType* indicates if the packet contains an EEM command message or an Ethernet frame, the bit *bCRC* specifies whether the CRC on the Ethernet frame has been calculated or set to a sentinel value ($0xdeadbeef$), and the remaining 14 bits are the length of the carried frame.

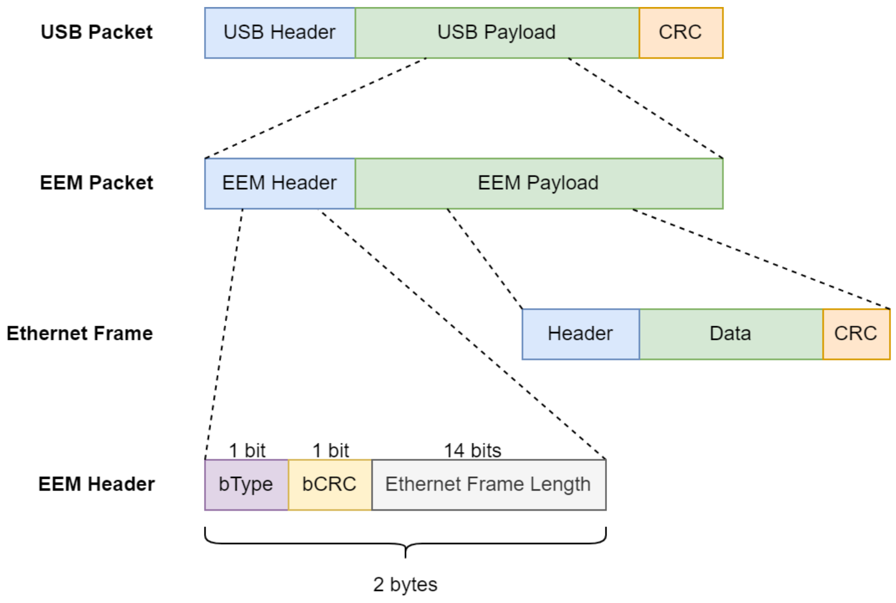


Figure 19: CDC-EEM packet layout and header contents.

When handling Ethernet Control Model packets, the task does not need to modify the header contents as ECM does not allow encapsulation, and the USB packet payload only contains the Ethernet frame.

4.3 Data Flow Management

When the Hub is attached to the Charger, the USB CDC-ECM enumeration process is done, making the Hub available to the network, with its own MAC and IP addresses. Henceforth, a data path is established between the Charger's Ethernet port and the Hub, via its USB interface.

There are three main components involved in the process of moving data across this path: the MCU's Ethernet peripheral (GMAC), the ARM CPU core (running application), and the MCU's USB module (USBHS), as Figure 20 illustrates. The USB and Ethernet tasks run independently within the Charger application. The GMAC is controlled by the Ethernet task and it is responsible for transfer operations on the Ethernet port. The USBHS is controlled by the USB task, governing the data transfers at the USB port. The Charger MCU software provides a bridge for Ethernet and USB peripherals, where data is moved between internal buffers as communication occurs between the LAN and the Hub.

This subsection will describe the data buffering mechanisms on the Charger MCU application and how the peripherals interact with the buffers.

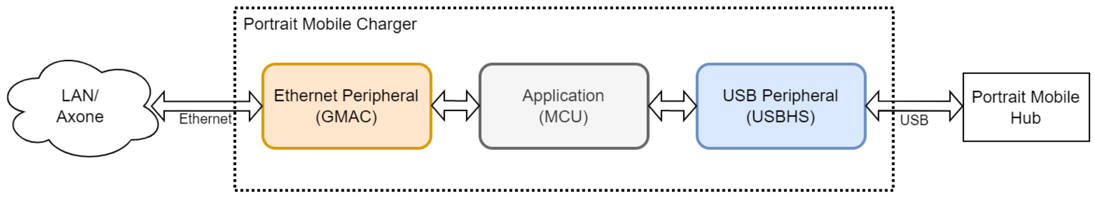


Figure 20: Data path for Ethernet communication between the Hub and Local Area Network via the Charger.

4.3.1 Charger Application Data Buffers

Barr and Massa suggest on [32] that, for extending the functionality of a serial communication and making the communication more robust, a FIFO buffering mechanism is recommended. Additionally, they mention that for embedded systems with more than one device driver, CDC-ACM and CDC-ECM in the Charger's case, it is important to consider how the limited MCU resources are used and shared between tasks. Taking this into consideration, two ring buffers were introduced to the software, one responsible for data moving from the USB peripheral (Hub) to the GMAC (network), named *gUsbRead*, and other for network data going into the Hub via the USBHS, *gUsbSend*. Both buffers are shared between the USB and Ethernet control tasks, which are running the device drivers to interface with their respective peripherals.

The buffers use a standard ring queue mechanism where data is written to a buffer on one end and read from the other. Figure 21 illustrates the two data ring buffers and the index trackers *gUsbSendEnqueuePos*, *gUsbSendDispatchPos*, *gUsbReadEnqueuePos*, and *gUsbReadDispatchPos*. The length of the ring buffers are static values configurable during software compilation, *m* for *gUsbRead* and *n* for *gUsbSend*. The index trackers are used to inform the drivers which position of the buffers data is being written or read. For instance, when a packet arrives from the network, the GMAC peripheral will process it and copy it to index *gUsbSendEnqueuePos* of *gUsbSend*, and increment the en-queue position by one. On the USB counterpart, the driver will check if index *gUsbSendDispatchPos* of *gUsbSend* contains data to be sent. If yes, the USB peripheral forwards the packet to the physical layer and the *gUsbSendDispatchPos* is incremented to the next position.

The Hub hardware is based on the i.MX7 platform with a multi-core processing unit and one gigabyte of LPDDR3 SDRAM, making it a much more capable hardware than the Charger's ATSAME70Q20, which has a single processing core and 384 kilobytes of SRAM. Subsequently, in a situation where the Hub pushes more USB packets than the Charger is able to read, this data will be temporarily stored on the Hub's internal RAM, backpressuring the flow and allowing the Charger to stall reading data until it has free memory on *gUsbRead*.

When packets arrive from the network, the Charger is entirely responsible for buffering until the Hub is able to consume it. If the GMAC finishes processing an incoming Ethernet packet and *gUsbSend* is full, that packet will simply be dropped.

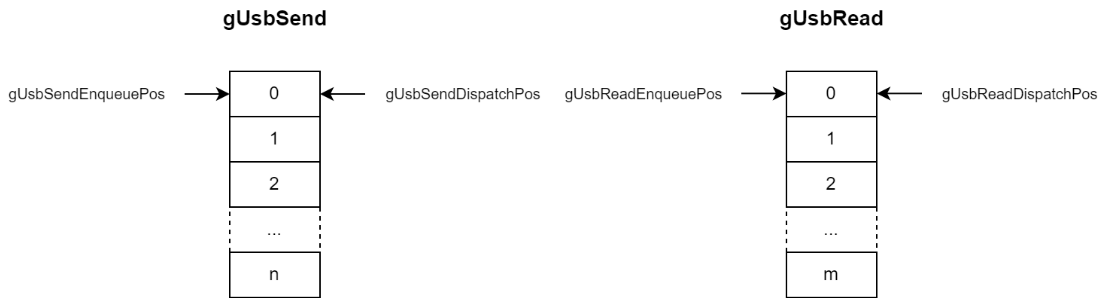


Figure 21: Ring buffers used for the Ethernet to USB data bridge and their index trackers.

The backpressure interaction with the Hub allows a larger memory allocation for the *gUsbSend* while still maintaining a balanced throughput on both directions. Through experimentation and based on memory availability on the Charger's software, it was found that a reasonable length for *gUsbSend* and *gUsbRead* is $n = 20$ and $m = 4$ buffers, respectively.

4.3.2 Buffer Sizes

Each buffer in the ring queue needs to be large enough to accommodate at least one maximum size IEEE 802.3 Ethernet frame (1536 bytes for the GMAC PHY configuration). The software supports both CDC-ECM and CDC-EEM protocols, therefore the extra overhead needed for packet encapsulation must be taken into consideration. The value chosen for buffer size is 1624 bytes, resulting in a total cache size of $1624 \times 20 = 32480$ bytes for downlink traffic (network to Hub), and $1624 \times 4 = 6496$ bytes for uplink (Hub to network).

4.3.3 Rate of Data Flow

As the GMAC fills the gUsbSend buffers, the USB peripheral is able to immediately consume the data, since, the bit rate of USB 2.0 is 480 Mbps, compared to the 100 Mbps link on the Ethernet port. However, the Ethernet peripheral can only consume data from the buffers at its own link speed, so the Ethernet device is the bottleneck when attempting to achieve the maximum transfer speeds.

Although data rates close to the maximum theoretical speeds were achieved, it can vary depending on the packet payload size. The design of the drivers allows only one Ethernet packet to be sent per USB frame (every 125 μs in HS mode), resulting in 8000 packets per seconds. With a packet size of 100 bytes, the maximum throughput becomes limited to 6.4 Mbps, but increasing the packet length to 1000 bytes, results in a proportional speed improvement to 64 Mbps. Maximum bit rates are obtained when packet size is set to the Ethernet frame limit of 1500 bytes, achieving approximately 94 Mbps.

The USB task design does not take advantage of the packet encapsulation feature of the EEM protocol, thus the throughput performance of CDC-ECM and CDC-EEM are equivalent. However, the extra overhead of header manipulation causes the CPU load on the ATSAME70Q20 to be slightly higher when the EEM configuration is selected. Consequently, it was decided to use the ECM configuration as the final solution for the Ethernet-USB bridge.

4.4 End-to-end Design

Figure 22 summarizes the complete end-to-end design of the USB to Ethernet data bridge. The Ethernet task has direct control over the GMAC driver and it is able to queue packets for transmission as soon as they become available on *gUsbRead* by receiving an interrupt notification from the CDC-ECM driver. When an incoming network packet becomes available on the GMAC FIFO, the Ethernet task is able to copy it to the next available *gUsbSend* buffer. There is no need for sending the USB driver a notification when a packet becomes available, as the driver checks for buffered packets on every SOF.

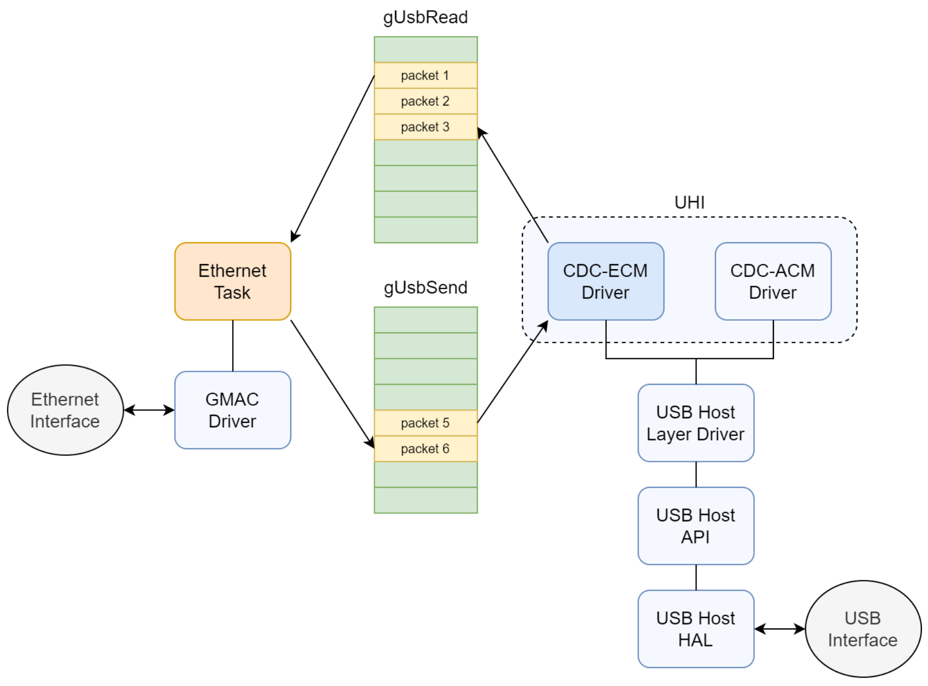


Figure 22: Ethernet to USB interface and its internal components.

Unlike the Ethernet task, the USB task does not have direct control over the USB transactions, but it is only able to start, suspend, and stop the driver operation. Once started, the driver operates autonomously via interrupts from the USBHS peripheral. This functionality allows passing the internal ring buffers addresses directly to the driver, which uses DMA to copy the data to/from the pipes. After every SOF, the USBHS triggers an interrupt to initiate receive and transmit operations at the configured pipes. Following the pipe priority, the transmission handler checks if there is a packet to be sent from *gUsbSend* and requests the peripheral to run this endpoint transaction. The reception handler is called in succession, requesting a read operation for the next available USB packet. Once the transfers are completed, a callback-function updates the ring buffers positional indexes. If all transactions were completed and there is still time within the USB frame before the next SOF, the driver will attempt to run another endpoint operation to maximize transfer speeds.

5 Testing Process

After the implementation of the software was completed, a series of tests were conducted to analyze the performance of the data bridge and evaluate the solution based on the defined requirements. This chapter describes the testing setup as well as the methods used to determine the performance and stability of the system. The data collected was used to analyze how well the Charger is able to transfer packets and in which situations losses start to occur.

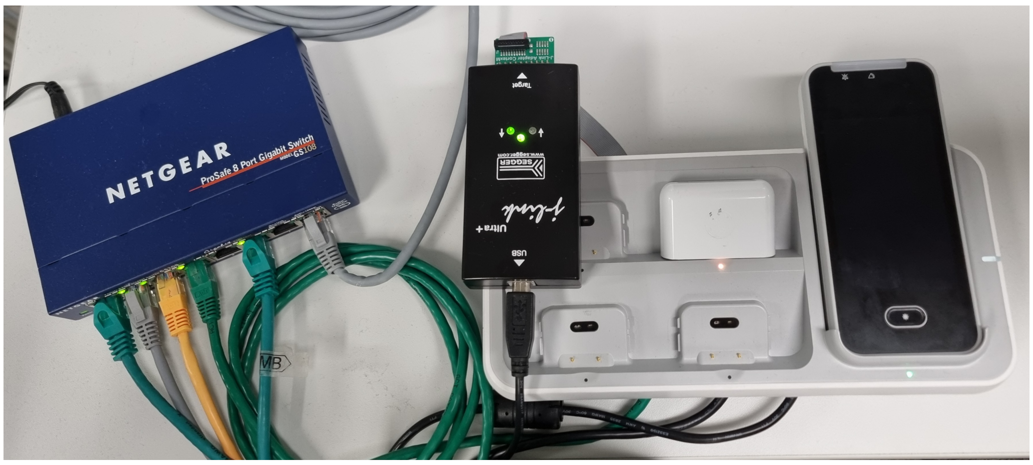


Figure 23: Setup for testing the communication between Hub and PC via Ethernet.

The test setup, depicted on Figure 23, includes the Charger, flashed with the software including the LAN functionality, a Hub and a Sensor Battery. The Charger and the test PC are connected to the LAN via a network switch. For debugging purposes and real time logging, a Segger J-Link probe is attached to the ATSAME70Q20's JTAG interface. Figure 24 shows a diagram including the devices used for the testing process and their connections.

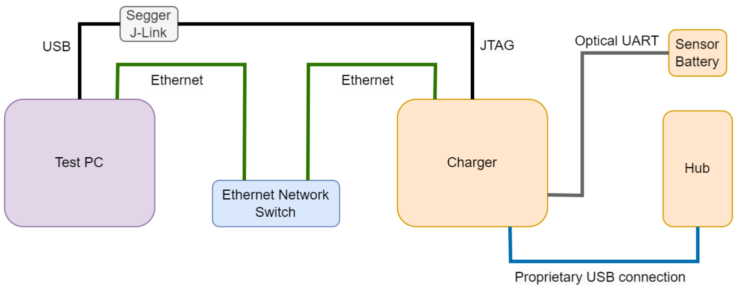


Figure 24: Diagram with the devices and connections used for the testing process.

The test PC runs a Linux operating system and it includes the open-source tools *iPerf* and *Ping*, used to measure network bandwidth and latency, respectively. In addition to these tools, two other internally-developed software are used for testing realistic use cases and the UART connectivity, namely *Network Stream Service* and *Hub MCU Utility*.

5.1 iPerf

iPerf is a widely used open-source network performance testing tool that provides capabilities for measuring network throughput, jitter, and packet loss. It works by using a client-server model, generating traffic in form of data streams between two nodes. It can be used to evaluate network bandwidth capacity and to identify network issues that may impact application performance [34]. iPerf supports a variety of protocols, including TCP/IP and UDP, as well as provides the option to run as either client or server, allowing measurements of bidirectional traffic. It also provides options for controlling test parameters, such as packet size, data to be transferred, and test duration [33].

5.1.1 Setup

iPerf is installed on both the PC and Hub. The LAN is established by a switch connecting the devices, and a WLAN connects the Hub wireless interface with the computer via a router. The test computer accesses the Hub via wireless SSH connection to avoid disturbances on the Ethernet communication.

- iPerf version 3.7
- Charger SW version *lan_charger_poc_7fbfe1a*
- Linux PC as iPerf3 client (192.168.8.3)
- Hub as iPerf3 server (192.168.8.20)
- SSH connection to the Hub via wireless interface (192.168.175.159)

5.1.2 TCP

In TCP, there is a flow control in place to determine how fast data can be sent without acknowledgement, the sliding window algorithm [35]. If a packet is lost, the sender will try to re-transmit it, facilitating a reliable and ordered data delivery. For this test, the bandwidth was measured under two scenarios: transfers based on total data size, and transfers with packet length set to a specific value. Table 6 describes the commands used on this test.

Table 6: iPerf3 commands used for testing the bandwidth between Hub and PC with Charger acting as the data bridge using the TCP protocol.

Command	**Description**
iperf3 -s 192.168.8.20 -V	Start iPerf3 server on the Hub on verbose mode.
iperf3 -c 192.168.8.20 -V -n <size>	Start client on the PC, flag **n** sets the amount of data to be transferred in bytes.
iperf3 -c 192.168.8.20 -V -l <length>	Start client on the PC, flag **l** sets the packet size in bytes of each transfer.

The tests were done in each direction, from PC to the Hub (downlink) and from Hub to PC (uplink). A third scenario where uplink and downlink are running simultaneously (bidirectional) is analyzed as well. The flag **-R** is used to perform the test on reverse mode (uplink), and the flag **-bidir** will run the test in bidirectional mode. For the first scenario, data size values of 100 MB and 500 MB were chosen. In the second scenario, packets sizes of 64 KB, 100 KB, 500 KB, and 1448 KB were used. Tables 7 and 8 summarize the results and the complete test logs can be found in Appendix A.1.

Table 7: iPerf average transfer speeds when setting fixed data sizes to be transmitted using TCP protocol.

Direction	**Data to Transfer**	**Average Speed**
Uplink	100 MB	93.6 Mbps
Uplink	500 MB	93.7 Mbps
Downlink	100 MB	91.4 Mbps
Downlink	500 MB	91.4 Mbps
Bidirectional	100 MB	Down 42.4 Mbps / Up 76.8 Mbps
Bidirectional	500 MB	Down 45.2 Mbps / Up 75.8 Mbps

Table 8: iPerf average transfer speeds when setting fixed packet lengths using the TCP protocol.

Direction	**Packet Length**	**Average Speed**
Uplink	64 bytes	18 Mbps
Uplink	100 bytes	27.7 Mbps
Uplink	500 bytes	90.3 Mbps
Uplink	1448 bytes	93.7 Mbps
Downlink	64 bytes	32.1 Mbps
Downlink	100 bytes	52.7 Mbps
Downlink	500 bytes	90.8 Mbps
Downlink	1448 bytes	90.9 Mbps
Bidirectional	64 bytes	Down 3.36 Mbps / Up 33.6 Mbps
Bidirectional	100 bytes	Down 3.81 Mbps / Up 38.1 Mbps
Bidirectional	500 bytes	Down 42.9 Mbps / Up 75.4 Mbps
Bidirectional	1448 bytes	Down 46.9 / Up 74.0 Mbps

5.1.3 UDP

Similarly to the TCP tests, iPerf was set to run in each direction, as well as in bidirectional mode. For each direction different packet lengths were selected, with values set for 64, 100, 500, and 1448 bytes. Since UDP does not have a flow control algorithm, packets are sent as fast as the sender desires, and any packet lost in the process is not re-transmitted. The quality of the network link can be determined by the amount of dropped packets for a given bandwidth. In this test, the bandwidth was set to the maximum value at which there is no longer packet loss. The maximum bandwidth so that packet loss would be minimal was found to be 4.1 Mbps, 6.4 Mbps, 32 Mbps, and 90 Mbps for packet sizes of 64 bytes, 100 bytes, 500 bytes, and 1448 bytes, respectively. Table 9 describes the commands used on this test.

Table 9: iPerf3 commands used for testing the bandwidth between Hub and PC with Charger acting as the data bridge using the UDP protocol.

Command	**Description**
iperf3 -s 192.168.8.20 -V	Start iPerf3 server on the Hub on verbose mode.
iperf3 -c 192.168.8.20 -u -b <bandwidth> -l <length> -V	Start client on the PC, flag **b** sets the target bandwidth, flag **u** sets the network protocol to UDP, and flag **l** sets the packet size.

Tests are done in each direction individually, as well as in bidirectional mode. The flags **-R** and **-bidir** are used for setting the traffic directions. Table 10 summarizes the test results with the packet loss for each bandwidth and packet length setting. The complete test logs can be found in Appendix A.2.

Table 10: Maximum achievable bandwidth while maintaining minimum packet loss using UDP protocol to transfer Ethernet frames through the Charger MCU.

Direction	**Packet Length**	**Bandwidth**	**Packet Loss**
Uplink	64 bytes	4.1 Mbps	0.12%
Uplink	100 bytes	6.4 Mbps	0.15%
Uplink	500 bytes	32 Mbps	0.033%
Uplink	1448 bytes	90 Mbps	0.084%
Downlink	64 bytes	4.1 Mbps	0%
Downlink	100 bytes	6.4 Mbps	0%
Downlink	500 bytes	32 Mbps	0%
Downlink	1448 bytes	90 Mbps	0%
Bidirectional	64 bytes	4.1 Mbps	Down 0.024% / Up 0%
Bidirectional	100 bytes	6.4 Mbps	Down 0.02% / Up 0%
Bidirectional	500 bytes	32 Mbps	Down 0.33% / Up 0%
Bidirectional	1448 bytes	90 Mbps	Down 1.4% / Up 0%

Given the design of the data flow allows a fixed 8000 packets per second to be transferred, the results are consistent with the expectations, since for packet lengths of

64, 100, 500, and 1448 bytes the bit rate would be 4.096 Mbps, 6.4 Mbps, 32 Mbps, and 92.672 Mbps, respectively. The exact values for packet loss varied for different runs, so should not be taken as an indication of performance, but rather to show the bandwidth is set to a value close to its maximum.

When the packet size is set to the Maximum Transfer Unit (MTU) of 1448 bytes while UDP traffic runs bidirectionally at full bandwidth, the limitations of the MCU and the data flow design can be seen. This scenario has a significantly higher packet loss in the downlink direction, as the Charger is not able to offload the *gUsbSend* buffers quickly enough, but the Hub is making use of its backpressuring capability to avoid dropping packets in the uplink direction.

5.2 Ping

Ping is an open-source utility tool used to test the reachability of network devices. It works by sending Internet Control Message Protocol (ICMP) echo request messages to the target device and waiting for a reply. If the device responds, the tool reports information about the transaction, such as response time, number of packets transmitted, and the amount of lost packets [36]. This is useful to determine network latency between two devices. A simple *ping* test using the default settings will send ICMP packets 64 bytes of length, and report the statistics of the round trip times of these packets. The version of *ping* used on the tests is from the collection of networking utilities *iputils* version *s20190709* for Linux.

Uplink and downlink latency where tested independently. For measuring uplink latency, *ping* was ran from the Hub, and for downlink, the *ping* was executed from the test PC. The tool can be started with the flag **w** to set how long, in seconds, the test should run for. Table 11 contains the commands used for testing network latency between the Hub and the PC.

Table 11: Ping commands used for measuring the latency between the Hub and the test PC in each direction.

Command	**Description**
ping 192.168.8.20 -w 10	Send ICMP ping request packets from the PC to the Hub
ping 192.168.8.3 -w 10	Send ICMP ping request packets from the Hub to the PC

The tests recorded an average of **0.892 ms** latency for uplink, and **0.944 ms** latency for downlink. The full results can be seen in Appendix B.

The *ping* tool offers the option for flooding a network port with packets and recording statistics on packet loss and round trip times. This can be done by using the flag **-f** and giving the application "superuser" rights with the *sudo* command. This feature was used to test the behaviour of the data link under flooding circumstances for different ICMP packet sizes, by setting the additional flag **-s**. Note that the ICMP header size is 8 bytes, so when setting the total packet size, the value for *-s* must be the desired size subtracted by eight. For example, for flooding the Hub interface with packets of size 500 bytes for 60 seconds, the command "*sudo ping 192.168.8.20 -s 492 -f -w 60*" can be used.

Flooding tests were performed during 60 seconds on both directions for packet sizes of 64, 100, 500 as well as 1448 bytes, and the results were recorded on Table 12.

Table 12: Ping results of a 60 second flooding test for different packet lengths and direction.

Direction	Packet Length	Packets Transmitted	Latency	Packet Loss
Uplink	64 bytes	66223	0.748 ms	0.002%
Uplink	100 bytes	63990	0.774 ms	0.002%
Uplink	500 bytes	56451	0.888 ms	0.002%
Uplink	1448 bytes	44269	1.158 ms	0%
Downlink	64 bytes	79582	0.707 ms	0.001%
Downlink	100 bytes	79292	0.710 ms	0.003%
Downlink	500 bytes	77670	0.720 ms	0.001%
Downlink	1448 bytes	47034	1.159 ms	0%

Larger packets take more time to be processed and transferred through the MCU. This can be verified by the higher average latency as the packet length increases. These tests generate a data rate well bellow the maximum achievable bandwidth of the data bridge, ranging between 0.5 Mbps and 9 Mbps, depending on packet size and direction. The small packet loss seen in these results are likely caused by signal interference on the data lines.

5.3 Network Stream Service

Network Stream Service (NSS) is a software tool developed by GE HealthCare to simulate realistic use cases and communication profiles between the Hub and Axone. The NSS supports the configuration of multiple data streams to simulate concurrent data traffic, for example, measurements from different sensors, trend information, and software updates. Each stream opens a UDP socket connection between the sender and receiver, with the option to configure transmissions with a burst pattern.

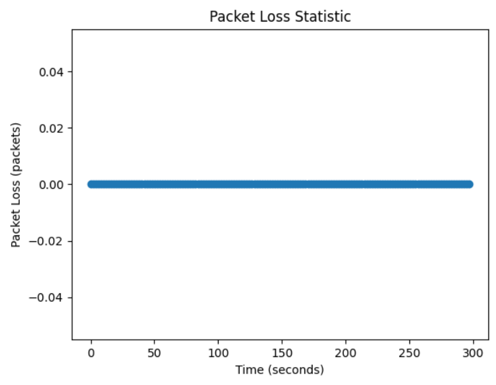
5.3.1 Patient Monitoring Simulation

The NSS test consists of a pair of sender-receiver, which can run independently on either the PC or Hub, depending on the desired direction of data flow. In a typical use case, the Hub would send periodic data to the Core Services including sensor data, and diagnostic trends. To simulate this behaviour, a data stream sender was placed on the Hub, with a corresponding data stream receiver running on the PC. Table 13 shows how the three streams were configured in this scenario, simulating SpO2 and respiration sensors sending measurements, as well as a trend update to indicate how the data changes over longer periods of time. When each stream sends data, the packets are sent in a burst at once.

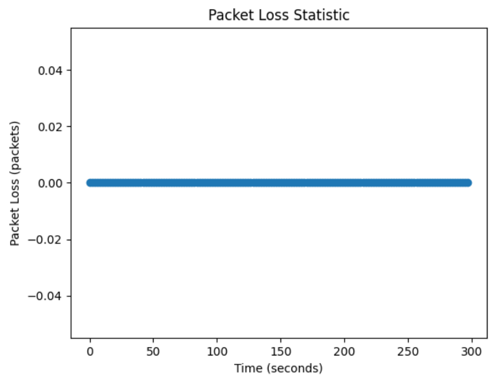
Table 13: Network Stream Service configuration for a realistic use case of patient monitoring.

Stream	Packet Size	Burst Size	Period	Delay Threshold
1	100 bytes	4 packets	200 ms	250 ms
2	400 bytes	2 packets	200 ms	250 ms
3	800 bytes	25 packets	5000 ms	5050 ms

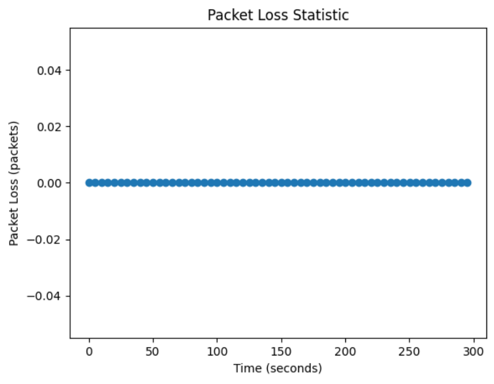
The test is active for 300 seconds and outputs diagnostic data about the streams, including how many packets were lost, and if there were any significantly delayed or out-of-order packets. The delay threshold value designates the maximum time difference allowed between when the packet was sent and received. Figures 25a, 25b and 25c show the plot of the amount of packet loss for equal subdivisions of time on stream one, two and three, respectively. There were no losses, delays or out-of-order packets on this simulation.



(a) Stream 1



(b) Stream 2



(c) Stream 3

Figure 25: Hub as sender: NSS packet loss on patient monitoring simulation for Streams 1, 2, and 3.

5.3.2 Stress Testing

A stress test was conducted to analyse how the Charger performs under heavy traffic load. For this scenario, the burst size of each stream was doubled compared to the nominal monitoring values, and incoming data was added to the Hub resulting in the stream configuration described in Tables 14 and 15.

Table 14: Hub Network Stream Service configuration for outgoing data in a stress test scenario.

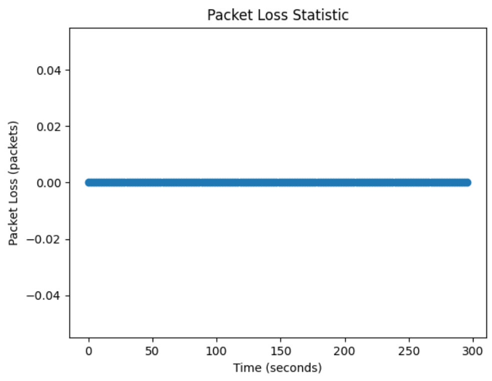
Stream	Packet Size	Burst Size	Period	Delay Threshold
1	100 bytes	8 packets	200 ms	250 ms
2	400 bytes	4 packets	200 ms	250 ms
3	800 bytes	50 packets	5000 ms	5050 ms

Table 15: PC Network Stream Service configuration for outgoing data in a stress test scenario.

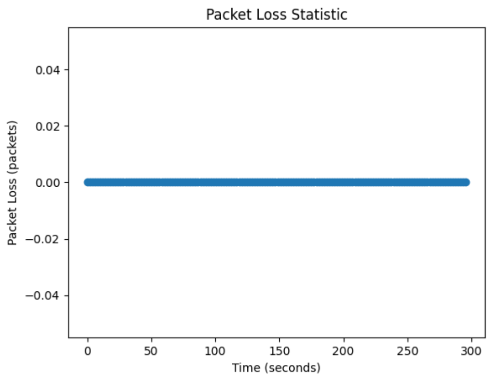
Stream	Packet Size	Burst Size	Period	Delay Threshold
4	100 bytes	8 packets	200 ms	250 ms
5	400 bytes	4 packets	200 ms	250 ms
6	800 bytes	50 packets	5000 ms	5050 ms
7	1400 bytes	30 packets	1000 ms	1500 ms

Analyzing the frames using the network capture tool Wireshark, the interval of packets sent in a burst can be determined. When packets are sent from the Hub, the inter-packet burst interval is 1 ms. When packets are sent from the PC, the interval is significantly shorter, being on average 100 μs.

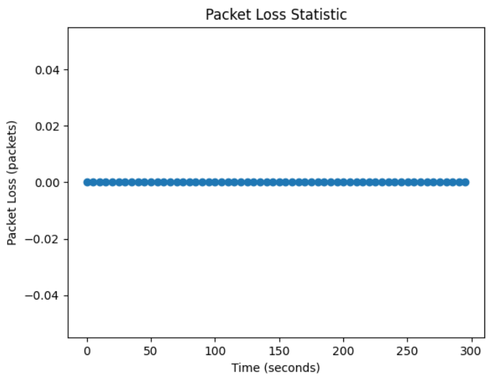
During the 300 seconds duration of the stress test, only 35 packets out of 51000 where lost, representing a **0.069% packet loss**, all of which were sent from the PC. Each data point on the graphs shown in Figures 26 and 27 represent the amount of lost packets in one data burst. *Stream 6* was the one with the most packet loss of 23, as Figure 27c shows. Figure 27a presents the loss statistics of *Stream 4*, with six lost packets. Six packet losses also occurred on *Stream 7*, as Figure 27d illustrates. *Stream 1*, *Stream 2*, *Stream 3*, and *Stream 5* had no packet losses, and their statistic graphs can be seen on Figures 26a, 26b, 26c, and 27b, respectively. There were no delayed or out-of-order packets.



(a) Stream 1



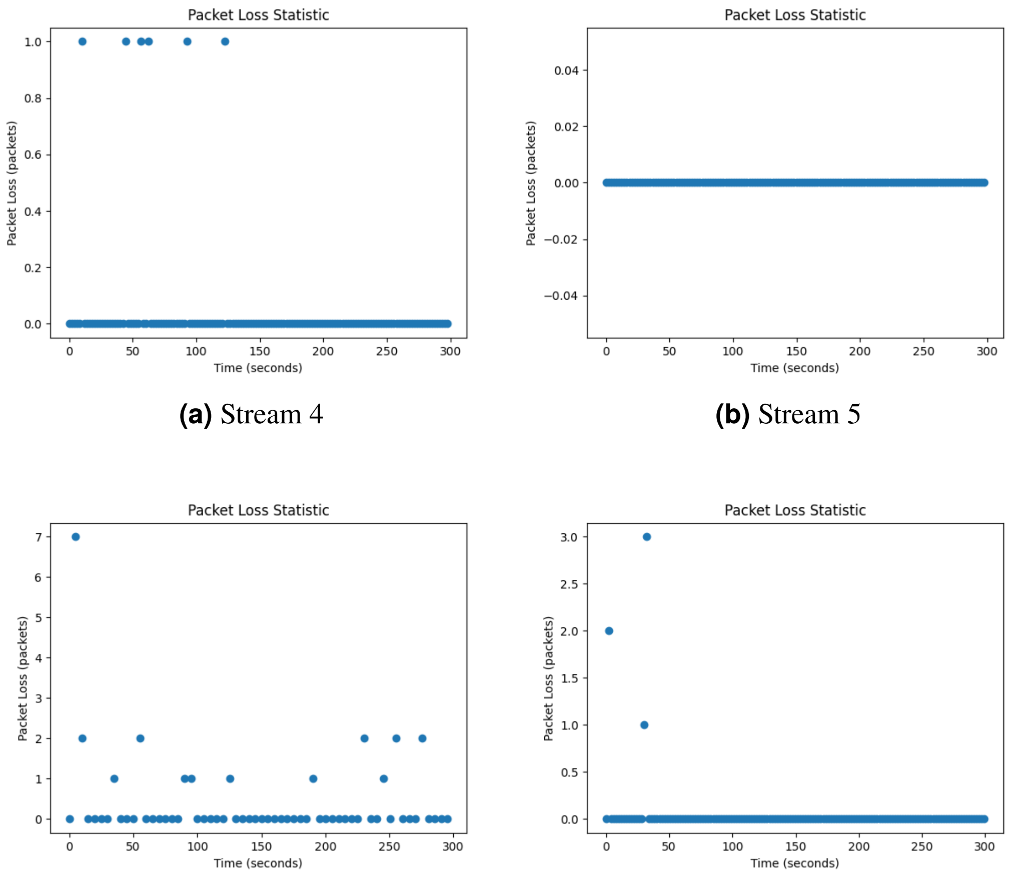
(b) Stream 2



(c) Stream 3

Figure 26: Hub as sender: NSS packet loss on stress test for Streams 1, 2, and 3.

The data streams with uplink direction did not experience any loss of data, as the Hub is able to backpressure the flow in case the *gUsbRead* buffers get full. However, packet loss was detected on the downlink data streams, as the Ethernet task does not have backpressure capability, dropping incoming packets if the *gUsbSend* buffers are full. The MCU was able to handle *Stream 5* with no loss, as it was the stream with least amount of packets per burst. *Stream 6* was the one with most losses, because of its large burst size. *Stream 4* and *Stream 7* had experienced occasional losses due to synchronization with other streams, which increases the resulting burst the MCU must handle.



(a) Stream 4

(b) Stream 5

(c) Stream 6

(d) Stream 7

Figure 27: PC as sender: NSS packet loss on stress test for Streams 4, 5, 6, and 7.

5.4 Hub MCU Utility

All components of the Portrait Mobile system share a common communication protocol based on UART. This is how sensors, batteries, Hub, Charger and service PC are able to transfer data between each other, that being monitoring measurements, diagnostics or software updates. The Hub MCU Utility tool was developed for testing and debugging purposes and it works by transmitting proprietary protocol messages by either the service PC (connected via USB to the Charger) or the Hub.

To test the performance of the UART emulation over USB, based on the CDC-ACM, the Hub MCU Utility tool is used from the Hub to exchange data with the Charger. It is possible to measure the UART communication speed and stability by transferring a file of fixed size between the devices. For downlink traffic, the file is copied from the Charger to the Hub, and for uplink, the same file is copied from the Hub to the Charger.

5.4.1 UART Downlink Using CDC-ACM

Hub MCU Utility was used on the Hub to transmit a file simulating a software update for the Charger. The file size is 160292 bytes of raw binary data, and it is kept constant for all test cases. The tool provides the option to set the block size of each packet for any value between 8 and 256 bytes. Measurements for the file transfer using different block sizes were done and recorded in Table 16.

Table 16: UART transfer speeds from Hub to Charger when using CDC-ACM emulation via USB.

Block Size (Bytes)	File Size (KB)	Time (s)	Throughput (Kbps)
8	160,292	220	5,8
16	160,292	111	11.6
32	160,292	56	22,9
64	160,292	28	45,8
96	160,292	20	64,1
128	160,292	23	55,8
160	160,292	18	71,2
256	160,292	11	116,6

It was observed that throughput scales linearly with block size until around 96 bytes, then it begins to increase at a slower rate. Note that, with block size 128 bytes, there is a drop in throughput due to a timing interference caused by the CDC-ECM pipes. For data going out of the Charger MCU, the higher priority CDC-ACM output pipe is able to serve the packet transfer requests without having to wait for any other pipe to execute or timeout.

5.4.2 UART Uplink Using CDC-ACM

In this test case, Hub MCU Utility runs from the PC, sending commands to the Charger to initiate a file transfer from its flash memory to the Hub. The file is the same as used for UART downlink test, and block sizes also ranging from 8 to 256 bytes. Table 17 includes the results of the test.

Table 17: UART transfer speeds from Charger to Hub when using CDC-ACM emulation via USB.

Block Size (Bytes)	File Size (KB)	Time (s)	Throughput (Kbps)
8	160,292	598	2,1
128	160,292	38	33,7
256	160,292	19	67,5

While uplink UART traffic is active (data flowing into the Charger), the CDC-ACM pipe is blocked completely by the CDC-ECM pipes, since it is the pipe with lowest priority on the USBHS peripheral. If there is no active Ethernet traffic, the USBHS will wait for the CDC-ECM input pipe to timeout, before giving priority to the next pipe. Thus every UART block transfer must wait for the timeout of the CDC-ECM input pipe of 30 ms. Since each block transfer takes a constant amount of time, the uplink throughput scales linearly with the data block size. If during the UART transfer a simultaneous Ethernet transfer begins, the ECM pipe will not stall ACM communication, and the 30 ms wait for each block transfer would be considerably reduced.

5.5 End-to-End Monitoring Test

The purpose of the end-to-end test is to use the Charger's LAN functionality on a complete Portrait Mobile monitoring environment, from sensor data acquisition to displaying the measurements on the Axone Viewer. Figure 28 shows the complete test setup: an SpO2 sensor paired to the Hub is measuring pulse and blood oxygenation; the Hub is docked to the Charger and it had its wireless interface shut down (so that only way to communicate to Axone is via Ethernet); the Charger is connected to the Axone network via Ethernet; and the monitor displays the patient data the Hub is sending to the network.

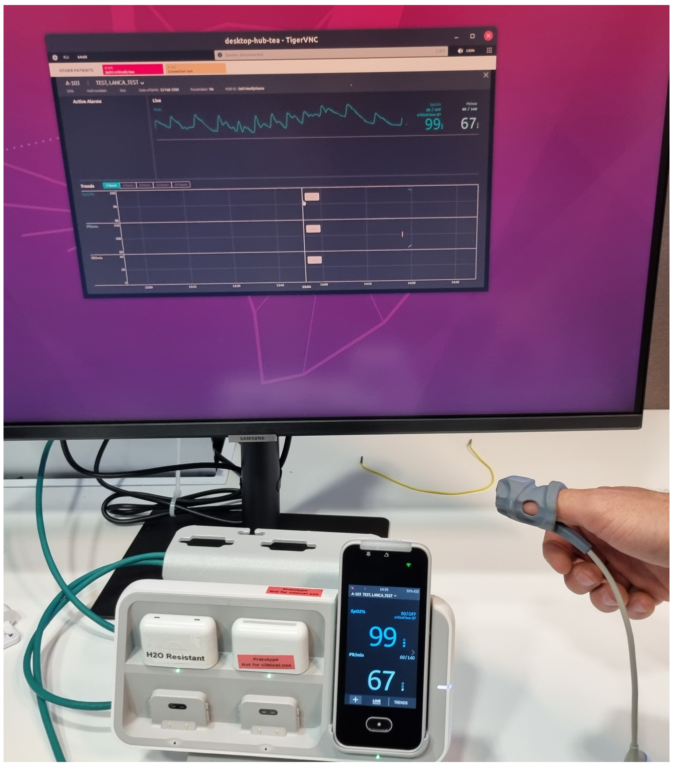


Figure 28: End-to-end test of Charger's LAN functionality.

The network topology of the system is presented in Figure 29. The sensors and the Hub communicate via a proprietary wireless network. The Hub wireless interface is disabled during the tests, so the access to the Axone server has to be made through the Charger's Ethernet interface. A laptop is connected to the Axone service via the WLAN router, gaining access to the data being monitored by the Hub.

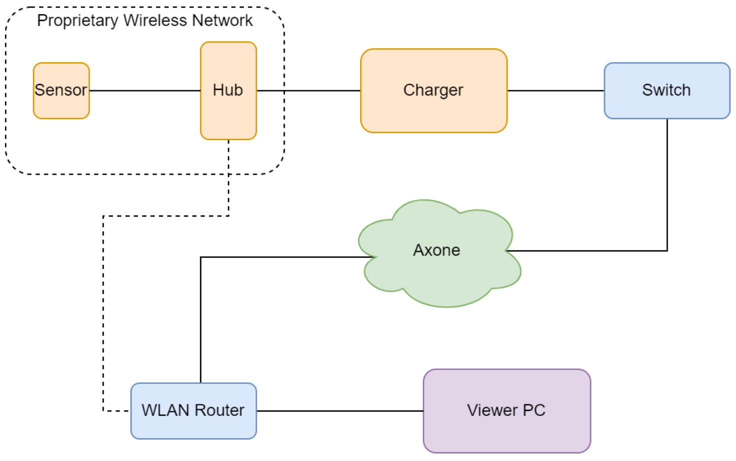


Figure 29: Network topology of the system for LAN functionality testing.

When undocked from the Charger, the Hub continues to display the sensor measurements, but the Viewer indicates a loss of connection from the patient, since the Ethernet connection was interrupted. As soon as the Hub is docked once again, the Viewer resumes displaying the SpO2 data. The re-connection process is seamless from the perspective of the user, as the Charger is able to reconfigure the USB multiplexer and do the enumeration effectively with no errors.

5.6 Results Analysis

The new implemented software was able to provide the Charger with USB host capabilities, enumerating the Hub as a CDC-ACM and CDC-ECM device, effectively enabling concurrent Ethernet and UART communication. This allows the Hub to use the Charger as a USB to Ethernet interface to transmit monitoring data to the network when docked. The performance evaluation using the tools *iPerf* and *ping* indicates the Charger is capable of providing a bandwidth over 90 Mbps, close to the rated value of the hardware (100 Mbps for GMAC), while maintaining sub-millisecond latency and minimal packet loss.

The Network Stream Service and end-to-end testing provided indications the Charger is capable of being used for the current and future monitoring needs, handling bursts of packets in multi-stream configurations with minimum data loss. The UART communication, mainly used for device diagnostics and software updates, was retained and verified by the *Hub MCU Utility* tests. New features and functionality were added to the Charger, including a data bridge for TCP/IP and UDP communication protocols, USB full-speed and high-speed mode support, as well as configurable Ethernet over USB models CDC-ECM and CDC-EEM.

One of the main challenges of the software implementation was to understand the interoperability of the USBHS peripheral. Because of the interrupt-driven functionality of the device, there is a lack of direct control over the USB functions, as all the data access and device enumeration process is done by interrupt handlers. The management of concurrent CDC-ACM and CDC-ECM traffic was done by careful selection of pipe allocation priority and timeout values. For this reason, and given the implemented buffering mechanism, there was a slight imbalance between uplink and downlink throughput.

6 Conclusion

The objective of this thesis was to provide a software solution for a high-speed communication interface between USB and Ethernet. The motivation was to solve a requirement for the new iteration of GE HealthCare's Portrait Mobile products, where the Hub would need a stable and fast way of communicating with the core services during patient monitoring, while being docked to the Charger. The presented solution was to use the Charger's software as a data bridge between the USB and Ethernet peripherals of the ATSAME70Q20 microcontroller, by implementing the support for Ethernet emulation protocols. A performance evaluation was done to conclude the software was able to bridge the communication between the Hub and the LAN in a fast and reliable manner.

This solution will contribute to the evolution of the Portrait Mobile products, by providing a high-bandwidth stable connection for the Hub communication, allowing the development of higher precision sensors and multi-parameter monitoring. The LAN connectivity adds an extra level of robustness to the monitoring system, potentially giving the patients and medical staff a higher confidence on the product. This implementation also helps to offload the wireless infrastructure on the hospitals, reducing interference and data loss.

The thesis had a higher focus on the USB operations, and most of the optimizations were done on that section of the data path. One of the limitations of this study was the Ethernet peripheral GMAC not being analyzed as much as the USBHS, as the hardware support for 10/100 Mbps caused a bottleneck on the data path. There are improvements to be done on the GMAC driver that could potentially increase the bandwidth for small sized packets and optimize CPU utilization. For future developments on micro-controllers that support gigabit Ethernet, improving the efficiency of the GMAC driver is highly recommended. Another limitation of this thesis was the simplistic approach for the CDC-EEM implementation. Although more taxing on CPU usage, CDC-EEM has the potential for achieving higher bandwidth than CDC-ECM, given its packet encapsulation capabilities. The Charger's software was designed to support both models, but the encapsulation method was not developed further. CDC-ECM was chosen as the final emulation model given its lower CPU usage and satisfactory transfer speeds.

The development of the USB to Ethernet interface inspired multiple topics to be further studied. Kim et al. methods for improving throughput on uses of CDC-EEM could greatly improve transfer speeds for small size packets [29]. The use of a lock-free Single-Producer Single-Consumer (SPSC) buffering mechanism would improve the performance and eliminate the risk of race conditions [37]. Exploring the possibility of implementing support for additional Ethernet emulation protocols, such as RNDIS and NCM, to provide more flexibility and compatibility with other types of devices. And investigate the use of similar micro-controllers that support multi-gigabit Ethernet and faster USB versions.

This thesis has made a number of contributions to the field of network communication, mainly by presenting the software design process for a high-speed communication interface between USB and Ethernet. The proposed testing methods and performance verification tools can be a useful reference for similar research topics. This work also demonstrated the feasibility of using a software solution for the data bridge, proving to be a valuable alternative to dedicated off-the-shelf hardware. Overall, the thesis has contributed to the field of USB and network communication by presenting a method for implementing a USB to Ethernet interface on an embedded device, as well as identifying the limitations and areas for improvement. This can guide future research and development in this area, leading to more efficient and reliable communication solutions.

References

[1] H. Chao, "Improving patient safety with RFID and mobile technology," *International Journal of Electronic Healthcare*, vol. 3, no. 2, pp. 175-192, April 2007. DOI: 10.1504/IJEH.2007.013099.

[2] D. M. Berwick, D. R. Calkins, C. J. McCannon, et al., "The 100,000 lives campaign: setting a goal and a deadline for improving health care quality," *JAMA*, vol. 295, no. 3, pp. 324-327, 2006. DOI: 10.1001/JAMA.295.3.324.

[3] K. K. Hall, A. Lim, and B. Gale. (Mar. 2020) *Making HealthCare Safer III: A Critical Analysis of Existing and Emerging Patient Safety Practices* [Online]. Available: https://www.ncbi.nlm.nih.gov/books/NBK555513/

[4] K. Kohtamäki, "Wearable technology innovations can revolutionise patient monitoring and recovery," [Online]. Available: https://www.vttresearch.com/en/news-and-ideas/wearable-technology-innovations-can-revolutionise-patient-monitoring-and-recovery [Accessed: 2022-09-05].

[5] USB Implementers Forum, "Universal Serial Bus Class Definitions for Communications Devices," CDC specification revision 1.2 [Online], 2010-10-3. Available: https://www.usb.org/document-library/class-definitions-communication-devices-12

[6] U.S. Food and Drug Administration, "Regulatory Controls," 2021. [Online]. Available: https://www.fda.gov/medical-devices/overview-device-regulation/regulatory-controls [Accessed: 2022-09-05]

[7] USB Implementers Forum, "Universal Serial Bus Specification," USB specification revision 2.0 [Online], 2000-4-27. Available: https://www.usb.org/document-library/usb-20-specification

[8] USB Implementers Forum, "USB Data Flow Model," in *Universal Serial Bus Specification*, revision 2.0, 2000-4-27, pp. 25-84.

[9] USB Implementers Forum, "USB Device Framework," in *Universal Serial Bus Specification*, revision 2.0, 2000-4-27, pp. 239-274.

[10] USB Implementers Forum, "Protocol Layer," in *Universal Serial Bus Specification*, revision 2.0, 2000-4-27, pp. 195-238.

[11] FTDI, "Simplified Description of USB Device Enumeration," Technical Note TN_113, Aug. 2020. [Online]. Available: https://ftdichip.com/wp-content/uploads/2020/08/TN_113_Simplified-Description-of-USB-Device-Enumeration.pdf. [Accessed: 2023-03-13].

[12] USB Implementers Forum, "Electrical," in *Universal Serial Bus Specification*, revision 2.0, 2000-4-27, pp. 119-194.

[13] USB Implementers Forum, *About USB-IF*, Available: https://www.usb.org/about. [Accessed: 2022-09-06]

[14] J. Corbet, A. Rubini, and G. Kroah-Hartman, *Linux Device Drivers*, 3rd ed. O'Reilly Media, 2005.

[15] R. B. Thompson, and B. F. Thompson, *PC Hardware in a Nutshell*, 3rd ed. O'Reilly Media, 2003.

[16] USB Made Simple. *A Series of Articles on USB* [Online]. Available: https://www.usbmadesimple.co.uk/ [Accessed 2022-09-20].

[17] Microchip Developer Guide. *How USB Works* [Online]. Available: https://microchipdeveloper.com/usb:how-it-works [Accessed 2022-09-21].

[18] Microchip Developer Guide. *Descriptor Tables* [Online]. Available: https://microchipdeveloper.com/usb:descriptor [Accessed 2022-09-21].

[19] J. Axelson, *USB Complete: Everything You Need to Develop Custom USB Peripherals*, 5th ed. Lakeview Research, 2015.

[20] USB Implementers Forum, "Universal Serial Bus Communications Class Subclass Specification for PSTN Devices," USB CDC-ACM specification revision 1.2 [Online], 2007-02-09. Available: https://usb.org/document-library/class-definitions-communication-devices-12

[21] USB Implementers Forum, "Universal Serial Bus Communications Class Subclass Specification for Ethernet Control Model Devices," USB CDC-ECM specification revision 1.2 [Online], 2007-02-09. Available: https://usb.org/document-library/class-definitions-communication-devices-12

[22] USB Implementers Forum, "Universal Serial Bus Communications Class Subclass Specification for Ethernet Emulation Model Devices," USB CDC-EEM specification revision 1.0 [Online], 2005-02-02. Available: https://usb.org/document-library/cdc-subclass-specification-ethernet-emulation-model-devices-10

[23] GE HealthCare. (2022). *Portrait Mobile* [Online]. Available: https://www.gehealthcare.co.uk/products/patient-monitoring/portrait-mobile [Accessed 2022-09-25]

[24] Microchip Technology Inc., "32-bit Arm Cortex-M7 MCUs with FPU, Audio and Graphics Interfaces, High-Speed USB, Ethernet, and Advanced Analog," SAM E70/S70/V70/V71 datasheet, Oct. 2013 [Revised July 2022].

[25] FreeRTOS. (2022). *The FreeRTOS Kernel* [Online]. Available: https://www.freertos.org/RTOS.html [Accessed 2022-11-08]

[26] R. Mikulak, R. McDermott, and M. Beauregard, *The Basics of FMEA*, 2nd ed. Productivity Press, 2008.

[27] Microchip Technology Inc., *Hi-Speed USB 2.0 to 10/100 Ethernet Controller LAN9500A* [Online]. Available: https://www.microchip.com/en-us/product/LAN9500A [Accessed 2023-03-12]

[28] A. Silberschatz, G. Gagne, and P. B. Galvin, "System Components," in *Operating System Concepts*, 9th ed. Wiley, 2013, pp. 838-861.

[29] K. Kim, J. Kim and A. Deep, "Throughput improvement for Ethernet over USB," *The 18th IEEE International Symposium on Consumer Electronics* (ISCE 2014), Jeju, Korea (South), 2014, pp. 1-2, DOI: 10.1109/ISCE.2014.6884363.

[30] R. Barry. (2016). *Mastering FreeRTOS Real Time Kernel, A Hands-On Tutorial Guide* [Online]. Available: https://freertos.org/Documentation/RTOS_book.html

[31] Yocto Project, *Linux Kernel Development Manual* [Online]. Available: https://docs.yoctoproject.org/1.6.1/kernel-dev/kernel-dev.html [Accessed 2023-03-11]

[32] M. Barr, and A. Massa, *Programming Embedded Systems: With C and GNU Development Tools*, 2nd ed. O'Reilly Media, 2006.

[33] J. Dugan, S. Elliott, B. A. Mah, J. Poskanzer, and K. Prabhu. *iPerf - The ultimate speed test tool for TCP, UDP and SCTP* [Online]. Available: https://iperf.fr/ [Accessed 2023-03-13]

[34] S. S. Kolahi, S. Narayan, D. D. T. Nguyen and Y. Sunarto, "Performance Monitoring of Various Network Traffic Generators," *2011 UkSim 13th International Conference on Computer Modelling and Simulation*, Cambridge, UK, 2011, pp. 501-506, DOI: 10.1109/UKSIM.2011.102.

[35] L. L. Peterson, and B. S. Davie, *Computer networks: A systems approach*, 5th ed. Morgan Kaufmann, 2012.

[36] Die.net, *Ping - Linux Manual Page* [Online]. Available: https://linux.die.net/man/8/ping [Accessed 2023-03-13]

[37] K. Kjeel. (2014). *Lock-Free Single-Producer Single-Consumer Circular Queue* [Online]. Available: https://www.codeproject.com/Articles/43510/Lock-Free-Single-Producer-Single-Consumer-Circular [Accessed 2023-03-22]

A iPerf Results

A.1 TCP

A.1.1 TCP - 500 MB - Uplink

```
iperf 3.7
Linux tms 5.13.0-40-generic #45~20.04.1-Ubuntu SMP Mon Apr 4 09:38:31 UTC 2022 x86_64
Time: Sat, 26 Nov 2022 13:43:40 GMT
Connecting to host 192.168.8.20, port 5201
      Cookie: hjl62xpht3ktgam2opagzha42nkf2dep2wc6
      TCP MSS: 1448 (default)
[  6] local 192.168.8.3 port 37408 connected to 192.168.8.20 port 5201
Starting Test: protocol: TCP, 1 streams, 131072 byte blocks, omitting 0 seconds,
524288000 bytes to send, tos 0
[ ID] Interval           Transfer     Bitrate         Retr  Cwnd
[  6]   0.00-1.00   sec  11.6 MBytes  97.6 Mbits/sec    6   53.7 KBytes
[  6]   1.00-2.00   sec  11.3 MBytes  94.4 Mbits/sec   25   46.7 KBytes
[  6]   2.00-3.00   sec  11.2 MBytes  93.9 Mbits/sec    3   73.5 KBytes
[  6]   3.00-4.00   sec  11.2 MBytes  93.7 Mbits/sec    2   69.3 KBytes
[  6]   4.00-5.00   sec  10.9 MBytes  91.7 Mbits/sec   49   60.8 KBytes
[  6]   5.00-6.00   sec  11.2 MBytes  94.3 Mbits/sec    5   73.5 KBytes
[  6]   6.00-7.00   sec  11.2 MBytes  94.0 Mbits/sec    3    100 KBytes
[  6]   7.00-8.00   sec  11.2 MBytes  93.7 Mbits/sec   55   86.3 KBytes
[  6]   8.00-9.00   sec  11.2 MBytes  94.2 Mbits/sec   13   59.4 KBytes
[  6]   9.00-10.00  sec  11.2 MBytes  94.3 Mbits/sec    6   58.0 KBytes
[  6]  10.00-11.00  sec  11.0 MBytes  92.0 Mbits/sec   44   70.7 KBytes
[  6]  11.00-12.00  sec  11.2 MBytes  93.9 Mbits/sec    5   53.7 KBytes
[  6]  12.00-13.00  sec  11.4 MBytes  95.6 Mbits/sec    2    107 KBytes
[  6]  13.00-14.00  sec  11.0 MBytes  91.9 Mbits/sec   29   59.4 KBytes
[  6]  14.00-15.00  sec  11.2 MBytes  93.7 Mbits/sec    4   67.9 KBytes
[  6]  15.00-16.00  sec  11.4 MBytes  95.6 Mbits/sec    7   86.3 KBytes
[  6]  16.00-17.00  sec  10.9 MBytes  91.7 Mbits/sec   33   90.5 KBytes
[  6]  17.00-18.00  sec  11.2 MBytes  93.7 Mbits/sec    4   70.7 KBytes
[  6]  18.00-19.00  sec  11.4 MBytes  95.6 Mbits/sec    3   94.7 KBytes
[  6]  19.00-20.00  sec  10.9 MBytes  91.7 Mbits/sec   24   77.8 KBytes
[  6]  20.00-21.00  sec  11.2 MBytes  93.7 Mbits/sec    6   56.6 KBytes
[  6]  21.00-22.00  sec  11.4 MBytes  95.6 Mbits/sec    5   69.3 KBytes
[  6]  22.00-23.00  sec  11.0 MBytes  92.0 Mbits/sec   20   60.8 KBytes
[  6]  23.00-24.00  sec  11.2 MBytes  94.0 Mbits/sec    5   82.0 KBytes
[  6]  24.00-25.00  sec  11.2 MBytes  94.0 Mbits/sec    6   66.5 KBytes
[  6]  25.00-26.00  sec  11.0 MBytes  92.1 Mbits/sec   31   49.5 KBytes
[  6]  26.00-27.00  sec  11.2 MBytes  94.0 Mbits/sec    9   59.4 KBytes
[  6]  27.00-28.00  sec  11.2 MBytes  94.0 Mbits/sec    5   83.4 KBytes
[  6]  28.00-29.00  sec  11.2 MBytes  93.9 Mbits/sec    8   38.2 KBytes
[  6]  29.00-30.00  sec  11.2 MBytes  93.7 Mbits/sec   17   59.4 KBytes
[  6]  30.00-31.00  sec  11.2 MBytes  93.7 Mbits/sec    5   67.9 KBytes
[  6]  31.00-32.00  sec  11.2 MBytes  93.9 Mbits/sec    4   74.9 KBytes
[  6]  32.00-33.00  sec  11.2 MBytes  93.7 Mbits/sec   39   87.7 KBytes
[  6]  33.00-34.00  sec  11.2 MBytes  93.9 Mbits/sec    9   67.9 KBytes
[  6]  34.00-35.00  sec  11.2 MBytes  94.0 Mbits/sec    2   99.0 KBytes
[  6]  35.00-36.00  sec  11.2 MBytes  93.7 Mbits/sec   57   66.5 KBytes
[  6]  36.00-37.00  sec  11.2 MBytes  94.0 Mbits/sec    4   69.3 KBytes
[  6]  37.00-38.00  sec  11.2 MBytes  93.8 Mbits/sec    7   39.6 KBytes
[  6]  38.00-39.00  sec  11.0 MBytes  91.9 Mbits/sec   22   66.5 KBytes
[  6]  39.00-40.00  sec  11.2 MBytes  93.8 Mbits/sec    9   48.1 KBytes
[  6]  40.00-41.00  sec  11.4 MBytes  95.6 Mbits/sec    2    116 KBytes
[  6]  41.00-42.00  sec  11.0 MBytes  92.1 Mbits/sec   52   52.3 KBytes
[  6]  42.00-43.00  sec  11.2 MBytes  93.9 Mbits/sec    5   60.8 KBytes
[  6]  43.00-44.00  sec  11.2 MBytes  93.7 Mbits/sec    4   82.0 KBytes
[  6]  44.00-44.73  sec  8.20 MBytes  94.4 Mbits/sec   28   63.6 KBytes
- - - - - - - - - - - - - - - - - - - - - - - - -
Test Complete. Summary Results:
[ ID] Interval           Transfer     Bitrate         Retr
[  6]   0.00-44.73  sec   500 MBytes  93.8 Mbits/sec  683             sender
[  6]   0.00-44.73  sec   499 MBytes  93.7 Mbits/sec                  receiver
```

```
CPU Utilization: local/sender 0.3% (0.0%u/0.3%s), remote/receiver 36.7% (1.7%u/35.0%s)
snd_tcp_congestion cubic
rcv_tcp_congestion cubic

iperf Done.
```

A.1.2 TCP - 100 MB - Uplink

```
iperf 3.7
Linux tms 5.13.0-40-generic #45~20.04.1-Ubuntu SMP Mon Apr 4 09:38:31 UTC 2022 x86_64
Time: Sat, 26 Nov 2022 13:42:59 GMT
Connecting to host 192.168.8.20, port 5201
      Cookie: 7ivdpoc3smmabxmty5m5v64tpp7ozcwbajfa
      TCP MSS: 1448 (default)
[  6] local 192.168.8.3 port 37404 connected to 192.168.8.20 port 5201
Starting Test: protocol: TCP, 1 streams, 131072 byte blocks, omitting 0 seconds,
104857600 bytes to send, tos 0
[ ID] Interval           Transfer     Bitrate         Retr  Cwnd
[  6]   0.00-1.00   sec  11.7 MBytes  97.8 Mbits/sec    3   60.8 KBytes
[  6]   1.00-2.00   sec  11.2 MBytes  94.0 Mbits/sec   28   63.6 KBytes
[  6]   2.00-3.00   sec  11.2 MBytes  93.7 Mbits/sec   11   42.4 KBytes
[  6]   3.00-4.00   sec  11.3 MBytes  94.5 Mbits/sec    5   73.5 KBytes
[  6]   4.00-5.00   sec  11.0 MBytes  92.0 Mbits/sec   46   63.6 KBytes
[  6]   5.00-6.00   sec  11.2 MBytes  93.6 Mbits/sec    5   91.9 KBytes
[  6]   6.00-7.00   sec  11.2 MBytes  93.7 Mbits/sec    4    102 KBytes
[  6]   7.00-8.00   sec  11.2 MBytes  94.2 Mbits/sec   40   99.0 KBytes
[  6]   8.00-8.92   sec  10.3 MBytes  94.2 Mbits/sec   12   50.9 KBytes
- - - - - - - - - - - - - - - - - - - - - - - - -
Test Complete. Summary Results:
[ ID] Interval           Transfer     Bitrate         Retr
[  6]   0.00-8.92   sec   100 MBytes  94.2 Mbits/sec  154             sender
[  6]   0.00-8.92   sec  99.5 MBytes  93.6 Mbits/sec                  receiver
CPU Utilization: local/sender 0.6% (0.0%u/0.5%s), remote/receiver 48.0% (1.3%u/46.7%s)
snd_tcp_congestion cubic
rcv_tcp_congestion cubic

iperf Done.
```

A.1.3 TCP - 500 MB - Downlink

```
iperf 3.7
Linux tms 5.13.0-40-generic #45~20.04.1-Ubuntu SMP Mon Apr 4 09:38:31 UTC 2022 x86_64
Time: Sat, 26 Nov 2022 13:54:51 GMT
Connecting to host 192.168.8.20, port 5201
Reverse mode, remote host 192.168.8.20 is sending
      Cookie: b4slutp7wxefd3b4k27f5kd7oddrpsnzrxnf
      TCP MSS: 1448 (default)
[  6] local 192.168.8.3 port 37426 connected to 192.168.8.20 port 5201
Starting Test: protocol: TCP, 1 streams, 131072 byte blocks, omitting 0 seconds,
524288000 bytes to send, tos 0
[ ID] Interval           Transfer     Bitrate
[  6]   0.00-1.00   sec  10.9 MBytes  91.1 Mbits/sec
[  6]   1.00-2.00   sec  10.9 MBytes  91.3 Mbits/sec
[  6]   2.00-3.00   sec  10.9 MBytes  91.0 Mbits/sec
[  6]   3.00-4.00   sec  10.9 MBytes  91.3 Mbits/sec
[  6]   4.00-5.00   sec  10.9 MBytes  91.4 Mbits/sec
[  6]   5.00-6.00   sec  10.9 MBytes  91.1 Mbits/sec
[  6]   6.00-7.00   sec  10.9 MBytes  91.6 Mbits/sec
[  6]   7.00-8.00   sec  10.9 MBytes  91.4 Mbits/sec
[  6]   8.00-9.00   sec  10.9 MBytes  91.1 Mbits/sec
[  6]   9.00-10.00  sec  10.9 MBytes  91.5 Mbits/sec
[  6]  10.00-11.00  sec  10.9 MBytes  91.3 Mbits/sec
[  6]  11.00-12.00  sec  10.9 MBytes  91.2 Mbits/sec
[  6]  12.00-13.00  sec  10.9 MBytes  91.4 Mbits/sec
[  6]  13.00-14.00  sec  10.9 MBytes  91.8 Mbits/sec
[  6]  14.00-15.00  sec  10.8 MBytes  90.8 Mbits/sec
[  6]  15.00-16.00  sec  10.9 MBytes  91.7 Mbits/sec
[  6]  16.00-17.00  sec  10.9 MBytes  91.6 Mbits/sec
[  6]  17.00-18.00  sec  10.9 MBytes  91.3 Mbits/sec
[  6]  18.00-19.00  sec  10.9 MBytes  91.5 Mbits/sec
[  6]  19.00-20.00  sec  10.8 MBytes  90.4 Mbits/sec
[  6]  20.00-21.00  sec  10.8 MBytes  90.8 Mbits/sec
[  6]  21.00-22.00  sec  10.9 MBytes  91.1 Mbits/sec
[  6]  22.00-23.00  sec  10.9 MBytes  91.8 Mbits/sec
[  6]  23.00-24.00  sec  10.9 MBytes  91.4 Mbits/sec
[  6]  24.00-25.00  sec  10.9 MBytes  91.4 Mbits/sec
[  6]  25.00-26.00  sec  10.9 MBytes  91.5 Mbits/sec
[  6]  26.00-27.00  sec  10.9 MBytes  91.4 Mbits/sec
[  6]  27.00-28.00  sec  10.9 MBytes  91.5 Mbits/sec
[  6]  28.00-29.00  sec  10.9 MBytes  91.5 Mbits/sec
[  6]  29.00-30.00  sec  10.9 MBytes  91.5 Mbits/sec
[  6]  30.00-31.00  sec  10.9 MBytes  91.8 Mbits/sec
[  6]  31.00-32.00  sec  10.9 MBytes  91.5 Mbits/sec
[  6]  32.00-33.00  sec  10.9 MBytes  91.6 Mbits/sec
[  6]  33.00-34.00  sec  10.9 MBytes  91.3 Mbits/sec
[  6]  34.00-35.00  sec  10.9 MBytes  91.1 Mbits/sec
[  6]  35.00-36.00  sec  10.8 MBytes  90.9 Mbits/sec
[  6]  36.00-37.00  sec  10.9 MBytes  91.4 Mbits/sec
[  6]  37.00-38.00  sec  10.9 MBytes  91.8 Mbits/sec
[  6]  38.00-39.00  sec  10.9 MBytes  91.5 Mbits/sec
[  6]  39.00-40.00  sec  10.9 MBytes  91.6 Mbits/sec
[  6]  40.00-41.00  sec  10.9 MBytes  91.8 Mbits/sec
[  6]  41.00-42.00  sec  10.9 MBytes  91.5 Mbits/sec
[  6]  42.00-43.00  sec  10.9 MBytes  91.3 Mbits/sec
[  6]  43.00-44.00  sec  10.9 MBytes  91.6 Mbits/sec
[  6]  44.00-44.58  sec  6.33 MBytes  91.7 Mbits/sec
- - - - - - - - - - - - - - - - - - - - - - - - -
Test Complete. Summary Results:
[ ID] Interval           Transfer     Bitrate
[  6]   0.00-44.58  sec  0.00 Bytes  0.00 bits/sec                  sender
[  6]   0.00-44.58  sec   486 MBytes  91.4 Mbits/sec                  receiver
rcv_tcp_congestion cubic
iperf3: interrupt - the client has terminated
```

A.1.4 TCP - 100 MB - Downlink

```
iperf 3.7
Linux tms 5.13.0-40-generic #45~20.04.1-Ubuntu SMP Mon Apr 4 09:38:31 UTC 2022 x86_64
Time: Sat, 26 Nov 2022 13:54:28 GMT
Connecting to host 192.168.8.20, port 5201
Reverse mode, remote host 192.168.8.20 is sending
      Cookie: ulrjgjlmx34jeew6jekbjyhcv6gevnk5mhrg
      TCP MSS: 1448 (default)
[  6] local 192.168.8.3 port 37420 connected to 192.168.8.20 port 5201
Starting Test: protocol: TCP, 1 streams, 131072 byte blocks, omitting 0 seconds,
104857600 bytes to send, tos 0
[ ID] Interval           Transfer     Bitrate
[  6]   0.00-1.00   sec  10.8 MBytes  91.0 Mbits/sec
[  6]   1.00-2.00   sec  10.9 MBytes  91.2 Mbits/sec
[  6]   2.00-3.00   sec  10.9 MBytes  91.5 Mbits/sec
[  6]   3.00-4.00   sec  10.8 MBytes  90.9 Mbits/sec
[  6]   4.00-5.00   sec  10.9 MBytes  91.8 Mbits/sec
[  6]   5.00-6.00   sec  10.9 MBytes  91.6 Mbits/sec
[  6]   6.00-7.00   sec  10.9 MBytes  91.5 Mbits/sec
[  6]   7.00-8.00   sec  10.9 MBytes  91.8 Mbits/sec
[  6]   8.00-9.00   sec  10.8 MBytes  90.9 Mbits/sec
[  6]   9.00-9.81   sec  8.81 MBytes  91.6 Mbits/sec
- - - - - - - - - - - - - - - - - - - - - - - - -
```

```
Test Complete. Summary Results:
[ ID] Interval           Transfer     Bitrate
[  6]   0.00-9.81   sec  0.00 Bytes  0.00 bits/sec                  sender
[  6]   0.00-9.81   sec   107 MBytes  91.4 Mbits/sec                  receiver
rcv_tcp_congestion cubic
iperf3: interrupt - the client has terminated
```

A.1.5 TCP - 500 MB - Bidirectional

```
iperf 3.7
Linux tms 5.13.0-40-generic #45~20.04.1-Ubuntu SMP Mon Apr 4 09:38:31 UTC 2022 x86_64
Time: Sat, 26 Nov 2022 15:18:04 GMT
Connecting to host 192.168.8.20, port 5201
      Cookie: pzax42igvt43rgb33a424azu3ejoarlf2346
      TCP MSS: 1448 (default)
[  6] local 192.168.8.3 port 37578 connected to 192.168.8.20 port 5201
[  8] local 192.168.8.3 port 37580 connected to 192.168.8.20 port 5201
Starting Test: protocol: TCP, 1 streams, 131072 byte blocks, omitting 0 seconds,
524288000 bytes to send, tos 0
[ ID][Role] Interval           Transfer     Bitrate         Retr  Cwnd
[  6][TX-C]   0.00-1.00   sec  5.77 MBytes  48.4 Mbits/sec    8   43.8 KBytes
[  8][RX-C]   0.00-1.00   sec  8.51 MBytes  71.4 Mbits/sec
[  6][TX-C]   1.00-2.00   sec  5.64 MBytes  47.3 Mbits/sec   11   31.1 KBytes
[  8][RX-C]   1.00-2.00   sec  8.92 MBytes  74.8 Mbits/sec
[  6][TX-C]   2.00-3.00   sec  5.33 MBytes  44.7 Mbits/sec    7   22.6 KBytes
[  8][RX-C]   2.00-3.00   sec  9.02 MBytes  75.7 Mbits/sec
[  6][TX-C]   3.00-4.00   sec  5.82 MBytes  48.8 Mbits/sec   10   36.8 KBytes
[  8][RX-C]   3.00-4.00   sec  8.99 MBytes  75.4 Mbits/sec
[  6][TX-C]   4.00-5.00   sec  5.65 MBytes  47.4 Mbits/sec    8   36.8 KBytes
[  8][RX-C]   4.00-5.00   sec  8.91 MBytes  74.7 Mbits/sec
[  6][TX-C]   5.00-6.00   sec  4.91 MBytes  41.2 Mbits/sec   13   35.4 KBytes
[  8][RX-C]   5.00-6.00   sec  9.14 MBytes  76.7 Mbits/sec
[  6][TX-C]   6.00-7.00   sec  5.35 MBytes  44.9 Mbits/sec    8   42.4 KBytes
[  8][RX-C]   6.00-7.00   sec  9.03 MBytes  75.8 Mbits/sec
[  6][TX-C]   7.00-8.00   sec  5.44 MBytes  45.6 Mbits/sec   10   31.1 KBytes
[  8][RX-C]   7.00-8.00   sec  9.08 MBytes  76.2 Mbits/sec
[  6][TX-C]   8.00-9.00   sec  5.39 MBytes  45.2 Mbits/sec   10   55.1 KBytes
[  8][RX-C]   8.00-9.00   sec  9.12 MBytes  76.5 Mbits/sec
[  6][TX-C]   9.00-10.00  sec  5.32 MBytes  44.6 Mbits/sec   12   35.4 KBytes
[  8][RX-C]   9.00-10.00  sec  9.09 MBytes  76.3 Mbits/sec
[  6][TX-C]  10.00-11.00  sec  5.57 MBytes  46.8 Mbits/sec    7   46.7 KBytes
[  8][RX-C]  10.00-11.00  sec  8.98 MBytes  75.3 Mbits/sec
[  6][TX-C]  11.00-12.00  sec  5.48 MBytes  46.0 Mbits/sec   13   36.8 KBytes
[  8][RX-C]  11.00-12.00  sec  9.06 MBytes  76.0 Mbits/sec
[  6][TX-C]  12.00-13.00  sec  5.23 MBytes  43.8 Mbits/sec   10   41.0 KBytes
[  8][RX-C]  12.00-13.00  sec  9.05 MBytes  76.0 Mbits/sec
[  6][TX-C]  13.00-14.00  sec  5.03 MBytes  42.2 Mbits/sec    8   35.4 KBytes
[  8][RX-C]  13.00-14.00  sec  9.08 MBytes  76.2 Mbits/sec
[  6][TX-C]  14.00-15.00  sec  4.80 MBytes  40.3 Mbits/sec   10   26.9 KBytes
[  8][RX-C]  14.00-15.00  sec  9.20 MBytes  77.1 Mbits/sec
[  6][TX-C]  15.00-16.00  sec  5.18 MBytes  43.5 Mbits/sec    7   36.8 KBytes
[  8][RX-C]  15.00-16.00  sec  9.15 MBytes  76.7 Mbits/sec
[  6][TX-C]  16.00-17.00  sec  5.57 MBytes  46.8 Mbits/sec   12   43.8 KBytes
[  8][RX-C]  16.00-17.00  sec  8.99 MBytes  75.4 Mbits/sec
[  6][TX-C]  17.00-18.00  sec  5.76 MBytes  48.3 Mbits/sec    9   42.4 KBytes
[  8][RX-C]  17.00-18.00  sec  9.00 MBytes  75.5 Mbits/sec
[  6][TX-C]  18.00-19.00  sec  5.62 MBytes  47.1 Mbits/sec   11   36.8 KBytes
[  8][RX-C]  18.00-19.00  sec  8.94 MBytes  75.0 Mbits/sec
[  6][TX-C]  19.00-20.00  sec  5.88 MBytes  49.3 Mbits/sec   10   48.1 KBytes
[  8][RX-C]  19.00-20.00  sec  8.89 MBytes  74.6 Mbits/sec
[  6][TX-C]  20.00-21.00  sec  5.15 MBytes  43.2 Mbits/sec   12   42.4 KBytes
[  8][RX-C]  20.00-21.00  sec  9.10 MBytes  76.3 Mbits/sec
[  6][TX-C]  21.00-22.00  sec  5.50 MBytes  46.1 Mbits/sec   10   43.8 KBytes
[  8][RX-C]  21.00-22.00  sec  9.09 MBytes  76.3 Mbits/sec
[  6][TX-C]  22.00-23.00  sec  6.08 MBytes  51.0 Mbits/sec    9   35.4 KBytes
[  8][RX-C]  22.00-23.00  sec  8.89 MBytes  74.6 Mbits/sec
[  6][TX-C]  23.00-24.00  sec  4.31 MBytes  36.2 Mbits/sec   10   45.2 KBytes
[  8][RX-C]  23.00-24.00  sec  9.43 MBytes  79.1 Mbits/sec
[  6][TX-C]  24.00-25.00  sec  4.72 MBytes  39.6 Mbits/sec    9   39.6 KBytes
[  8][RX-C]  24.00-25.00  sec  9.27 MBytes  77.7 Mbits/sec
[  6][TX-C]  25.00-26.00  sec  5.03 MBytes  42.2 Mbits/sec    9   29.7 KBytes
[  8][RX-C]  25.00-26.00  sec  9.19 MBytes  77.1 Mbits/sec
[  6][TX-C]  26.00-27.00  sec  4.59 MBytes  38.5 Mbits/sec    9   26.9 KBytes
[  8][RX-C]  26.00-27.00  sec  9.32 MBytes  78.1 Mbits/sec
[  6][TX-C]  27.00-28.00  sec  4.57 MBytes  38.4 Mbits/sec    5   45.2 KBytes
[  8][RX-C]  27.00-28.00  sec  9.35 MBytes  78.4 Mbits/sec
[  6][TX-C]  28.00-29.00  sec  5.12 MBytes  42.9 Mbits/sec   20   31.1 KBytes
[  8][RX-C]  28.00-29.00  sec  9.21 MBytes  77.2 Mbits/sec
[  6][TX-C]  29.00-30.00  sec  4.31 MBytes  36.2 Mbits/sec    5   41.0 KBytes
[  8][RX-C]  29.00-30.00  sec  9.44 MBytes  79.2 Mbits/sec
[  6][TX-C]  30.00-31.00  sec  5.41 MBytes  45.4 Mbits/sec   11   42.4 KBytes
[  8][RX-C]  30.00-31.00  sec  9.13 MBytes  76.6 Mbits/sec
[  6][TX-C]  31.00-32.00  sec  5.10 MBytes  42.8 Mbits/sec    9   48.1 KBytes
[  8][RX-C]  31.00-32.00  sec  9.22 MBytes  77.4 Mbits/sec
[  6][TX-C]  32.00-33.00  sec  4.68 MBytes  39.2 Mbits/sec    9   43.8 KBytes
[  8][RX-C]  32.00-33.00  sec  9.24 MBytes  77.5 Mbits/sec
[  6][TX-C]  33.00-34.00  sec  4.74 MBytes  39.8 Mbits/sec   10   50.9 KBytes
[  8][RX-C]  33.00-34.00  sec  9.22 MBytes  77.4 Mbits/sec
[  6][TX-C]  34.00-35.00  sec  4.89 MBytes  41.0 Mbits/sec   10   39.6 KBytes
[  8][RX-C]  34.00-35.00  sec  9.26 MBytes  77.6 Mbits/sec
[  6][TX-C]  35.00-36.00  sec  4.51 MBytes  37.8 Mbits/sec   10   38.2 KBytes
[  8][RX-C]  35.00-36.00  sec  9.34 MBytes  78.4 Mbits/sec
[  6][TX-C]  36.00-37.00  sec  4.89 MBytes  41.0 Mbits/sec    9   31.1 KBytes
[  8][RX-C]  36.00-37.00  sec  9.22 MBytes  77.4 Mbits/sec
[  6][TX-C]  37.00-38.00  sec  4.74 MBytes  39.8 Mbits/sec    7   45.2 KBytes
[  8][RX-C]  37.00-38.00  sec  9.25 MBytes  77.6 Mbits/sec
[  6][TX-C]  38.00-39.00  sec  5.36 MBytes  45.0 Mbits/sec   10   43.8 KBytes
[  8][RX-C]  38.00-39.00  sec  9.11 MBytes  76.5 Mbits/sec
[  6][TX-C]  39.00-40.00  sec  4.97 MBytes  41.7 Mbits/sec    7   48.1 KBytes
[  8][RX-C]  39.00-40.00  sec  9.14 MBytes  76.7 Mbits/sec
[  6][TX-C]  40.00-41.00  sec  5.24 MBytes  44.0 Mbits/sec   15   45.2 KBytes
[  8][RX-C]  40.00-41.00  sec  9.11 MBytes  76.4 Mbits/sec
[  6][TX-C]  41.00-42.00  sec  4.77 MBytes  40.0 Mbits/sec   11   32.5 KBytes
[  8][RX-C]  41.00-42.00  sec  9.24 MBytes  77.5 Mbits/sec
[  6][TX-C]  42.00-43.00  sec  5.94 MBytes  49.8 Mbits/sec   10   43.8 KBytes
[  8][RX-C]  42.00-43.00  sec  9.02 MBytes  75.7 Mbits/sec
[  6][TX-C]  43.00-44.00  sec  5.10 MBytes  42.8 Mbits/sec   11   41.0 KBytes
[  8][RX-C]  43.00-44.00  sec  9.16 MBytes  76.8 Mbits/sec
[  6][TX-C]  44.00-45.00  sec  5.10 MBytes  42.8 Mbits/sec    7   39.6 KBytes
[  8][RX-C]  44.00-45.00  sec  9.13 MBytes  76.5 Mbits/sec
[  6][TX-C]  45.00-46.00  sec  4.47 MBytes  37.5 Mbits/sec    8   29.7 KBytes
[  8][RX-C]  45.00-46.00  sec  9.33 MBytes  78.3 Mbits/sec
[  6][TX-C]  46.00-47.00  sec  4.57 MBytes  38.4 Mbits/sec   11   26.9 KBytes
[  8][RX-C]  46.00-47.00  sec  9.32 MBytes  78.2 Mbits/sec
[  6][TX-C]  47.00-48.00  sec  4.80 MBytes  40.3 Mbits/sec   10   31.1 KBytes
[  8][RX-C]  47.00-48.00  sec  9.06 MBytes  76.0 Mbits/sec
[  6][TX-C]  48.00-49.00  sec  4.65 MBytes  39.0 Mbits/sec    8   36.8 KBytes
[  8][RX-C]  48.00-49.00  sec  9.23 MBytes  77.4 Mbits/sec
[  6][TX-C]  49.00-50.00  sec  5.36 MBytes  45.0 Mbits/sec    6   55.1 KBytes
[  8][RX-C]  49.00-50.00  sec  9.05 MBytes  75.9 Mbits/sec
[  6][TX-C]  50.00-51.00  sec  5.03 MBytes  42.2 Mbits/sec   12   29.7 KBytes
[  8][RX-C]  50.00-51.00  sec  9.15 MBytes  76.7 Mbits/sec
[  6][TX-C]  51.00-52.00  sec  4.74 MBytes  39.8 Mbits/sec   12   35.4 KBytes
[  8][RX-C]  51.00-52.00  sec  9.29 MBytes  77.9 Mbits/sec
[  6][TX-C]  52.00-53.00  sec  4.94 MBytes  41.4 Mbits/sec   10   33.9 KBytes
[  8][RX-C]  52.00-53.00  sec  9.23 MBytes  77.4 Mbits/sec
[  6][TX-C]  53.00-54.00  sec  5.26 MBytes  44.1 Mbits/sec    9   35.4 KBytes
[  8][RX-C]  53.00-54.00  sec  9.09 MBytes  76.2 Mbits/sec
[  6][TX-C]  54.00-55.00  sec  4.68 MBytes  39.2 Mbits/sec    9   32.5 KBytes
[  8][RX-C]  54.00-55.00  sec  9.27 MBytes  77.8 Mbits/sec
[  6][TX-C]  55.00-56.00  sec  5.10 MBytes  42.8 Mbits/sec    7   39.6 KBytes
[  8][RX-C]  55.00-56.00  sec  9.17 MBytes  76.9 Mbits/sec
[  6][TX-C]  56.00-57.00  sec  4.97 MBytes  41.7 Mbits/sec   10   31.1 KBytes
[  8][RX-C]  56.00-57.00  sec  9.22 MBytes  77.4 Mbits/sec
[  6][TX-C]  57.00-58.00  sec  4.92 MBytes  41.3 Mbits/sec    7   32.5 KBytes
[  8][RX-C]  57.00-58.00  sec  9.17 MBytes  76.9 Mbits/sec
[  6][TX-C]  58.00-59.00  sec  4.45 MBytes  37.3 Mbits/sec    7   38.2 KBytes
[  8][RX-C]  58.00-59.00  sec  9.42 MBytes  79.0 Mbits/sec
[  6][TX-C]  59.00-60.00  sec  5.38 MBytes  45.1 Mbits/sec   12   33.9 KBytes
[  8][RX-C]  59.00-60.00  sec  9.08 MBytes  76.2 Mbits/sec
[  6][TX-C]  60.00-61.00  sec  5.32 MBytes  44.6 Mbits/sec    6   35.4 KBytes
[  8][RX-C]  60.00-61.00  sec  9.11 MBytes  76.4 Mbits/sec
[  6][TX-C]  61.00-62.00  sec  5.12 MBytes  42.9 Mbits/sec    6   39.6 KBytes
[  8][RX-C]  61.00-62.00  sec  9.16 MBytes  76.8 Mbits/sec
[  6][TX-C]  62.00-63.00  sec  4.48 MBytes  37.6 Mbits/sec    7   43.8 KBytes
[  8][RX-C]  62.00-63.00  sec  9.30 MBytes  78.0 Mbits/sec
[  6][TX-C]  63.00-64.00  sec  4.91 MBytes  41.2 Mbits/sec    9   36.8 KBytes
[  8][RX-C]  63.00-64.00  sec  9.18 MBytes  77.0 Mbits/sec
[  6][TX-C]  64.00-65.00  sec  4.92 MBytes  41.3 Mbits/sec    6   48.1 KBytes
[  8][RX-C]  64.00-65.00  sec  9.18 MBytes  77.0 Mbits/sec
[  6][TX-C]  65.00-66.00  sec  5.32 MBytes  44.6 Mbits/sec    6   46.7 KBytes
[  8][RX-C]  65.00-66.00  sec  9.11 MBytes  76.4 Mbits/sec
[  6][TX-C]  66.00-67.00  sec  4.77 MBytes  40.0 Mbits/sec   12   39.6 KBytes
[  8][RX-C]  66.00-67.00  sec  9.17 MBytes  77.0 Mbits/sec
[  6][TX-C]  67.00-68.00  sec  5.39 MBytes  45.2 Mbits/sec   13   39.6 KBytes
[  8][RX-C]  67.00-68.00  sec  9.09 MBytes  76.2 Mbits/sec
[  6][TX-C]  68.00-69.00  sec  4.54 MBytes  38.1 Mbits/sec   10   29.7 KBytes
[  8][RX-C]  68.00-69.00  sec  9.24 MBytes  77.5 Mbits/sec
[  6][TX-C]  69.00-70.00  sec  4.91 MBytes  41.2 Mbits/sec    5   50.9 KBytes
[  8][RX-C]  69.00-70.00  sec  9.27 MBytes  77.8 Mbits/sec
[  6][TX-C]  70.00-71.00  sec  5.32 MBytes  44.6 Mbits/sec   11   55.1 KBytes
[  8][RX-C]  70.00-71.00  sec  9.10 MBytes  76.3 Mbits/sec
[  6][TX-C]  71.00-72.00  sec  4.72 MBytes  39.6 Mbits/sec   14   49.5 KBytes
[  8][RX-C]  71.00-72.00  sec  9.26 MBytes  77.7 Mbits/sec
[  6][TX-C]  72.00-73.00  sec  4.44 MBytes  37.2 Mbits/sec    8   38.2 KBytes
[  8][RX-C]  72.00-73.00  sec  9.40 MBytes  78.8 Mbits/sec
[  6][TX-C]  73.00-74.00  sec  4.74 MBytes  39.8 Mbits/sec    9   36.8 KBytes
[  8][RX-C]  73.00-74.00  sec  9.23 MBytes  77.4 Mbits/sec
[  6][TX-C]  74.00-75.00  sec  5.62 MBytes  47.1 Mbits/sec    9   43.8 KBytes
[  8][RX-C]  74.00-75.00  sec  9.08 MBytes  76.2 Mbits/sec
[  6][TX-C]  75.00-76.00  sec  4.88 MBytes  40.9 Mbits/sec    7   50.9 KBytes
[  8][RX-C]  75.00-76.00  sec  9.21 MBytes  77.2 Mbits/sec
[  6][TX-C]  76.00-77.00  sec  5.57 MBytes  46.8 Mbits/sec    9   49.5 KBytes
[  8][RX-C]  76.00-77.00  sec  9.07 MBytes  76.1 Mbits/sec
[  6][TX-C]  77.00-78.00  sec  4.95 MBytes  41.5 Mbits/sec    8   42.4 KBytes
[  8][RX-C]  77.00-78.00  sec  9.24 MBytes  77.5 Mbits/sec
[  6][TX-C]  78.00-79.00  sec  5.24 MBytes  44.0 Mbits/sec    7   46.7 KBytes
[  8][RX-C]  78.00-79.00  sec  9.11 MBytes  76.4 Mbits/sec
[  6][TX-C]  79.00-80.00  sec  5.48 MBytes  46.0 Mbits/sec    9   45.2 KBytes
[  8][RX-C]  79.00-80.00  sec  9.03 MBytes  75.8 Mbits/sec
[  6][TX-C]  80.00-81.00  sec  5.45 MBytes  45.7 Mbits/sec    7   46.7 KBytes
[  8][RX-C]  80.00-81.00  sec  9.11 MBytes  76.4 Mbits/sec
[  6][TX-C]  81.00-82.00  sec  4.92 MBytes  41.3 Mbits/sec    8   33.9 KBytes
[  8][RX-C]  81.00-82.00  sec  9.17 MBytes  76.9 Mbits/sec
[  6][TX-C]  82.00-83.00  sec  5.01 MBytes  42.0 Mbits/sec    8   42.4 KBytes
[  8][RX-C]  82.00-83.00  sec  9.20 MBytes  77.1 Mbits/sec
[  6][TX-C]  83.00-84.00  sec  4.47 MBytes  37.5 Mbits/sec   15   46.7 KBytes
[  8][RX-C]  83.00-84.00  sec  9.35 MBytes  78.4 Mbits/sec
[  6][TX-C]  84.00-85.00  sec  5.44 MBytes  45.6 Mbits/sec    8   43.8 KBytes
[  8][RX-C]  84.00-85.00  sec  9.05 MBytes  76.0 Mbits/sec
[  6][TX-C]  85.00-86.00  sec  4.63 MBytes  38.9 Mbits/sec   10   41.0 KBytes
[  8][RX-C]  85.00-86.00  sec  9.27 MBytes  77.8 Mbits/sec
[  6][TX-C]  86.00-87.00  sec  4.94 MBytes  41.4 Mbits/sec    7   46.7 KBytes
[  8][RX-C]  86.00-87.00  sec  9.18 MBytes  77.0 Mbits/sec
[  6][TX-C]  87.00-88.00  sec  5.44 MBytes  45.6 Mbits/sec   16   29.7 KBytes
[  8][RX-C]  87.00-88.00  sec  9.09 MBytes  76.3 Mbits/sec
[  6][TX-C]  88.00-89.00  sec  4.37 MBytes  36.7 Mbits/sec   12   29.7 KBytes
[  8][RX-C]  88.00-89.00  sec  9.40 MBytes  78.9 Mbits/sec
[  6][TX-C]  89.00-90.00  sec  4.71 MBytes  39.5 Mbits/sec   11   38.2 KBytes
[  8][RX-C]  89.00-90.00  sec  9.26 MBytes  77.7 Mbits/sec
[  6][TX-C]  90.00-91.00  sec  5.10 MBytes  42.8 Mbits/sec    8   26.9 KBytes
[  8][RX-C]  90.00-91.00  sec  9.09 MBytes  76.2 Mbits/sec
[  6][TX-C]  91.00-92.00  sec  4.83 MBytes  40.5 Mbits/sec   12   25.5 KBytes
[  8][RX-C]  91.00-92.00  sec  9.23 MBytes  77.5 Mbits/sec
[  6][TX-C]  92.00-93.00  sec  4.62 MBytes  38.7 Mbits/sec    8   35.4 KBytes
[  8][RX-C]  92.00-93.00  sec  9.33 MBytes  78.3 Mbits/sec
[  6][TX-C]  93.00-94.00  sec  4.01 MBytes  33.6 Mbits/sec   10   39.6 KBytes
[  8][RX-C]  93.00-94.00  sec  9.50 MBytes  79.7 Mbits/sec
[  6][TX-C]  94.00-95.00  sec  4.95 MBytes  41.5 Mbits/sec   12   33.9 KBytes
[  8][RX-C]  94.00-95.00  sec  9.12 MBytes  76.5 Mbits/sec
[  6][TX-C]  95.00-96.00  sec  5.10 MBytes  42.8 Mbits/sec    9   46.7 KBytes
[  8][RX-C]  95.00-96.00  sec  9.19 MBytes  77.1 Mbits/sec
[  6][TX-C]  96.00-97.00  sec  5.16 MBytes  43.3 Mbits/sec    8   42.4 KBytes
[  8][RX-C]  96.00-97.00  sec  9.19 MBytes  77.1 Mbits/sec
[  6][TX-C]  97.00-98.00  sec  4.77 MBytes  40.0 Mbits/sec    9   42.4 KBytes
[  8][RX-C]  97.00-98.00  sec  9.24 MBytes  77.5 Mbits/sec
[  6][TX-C]  98.00-98.81  sec  4.57 MBytes  47.3 Mbits/sec    8   42.4 KBytes
[  8][RX-C]  98.00-98.81  sec  7.38 MBytes  76.2 Mbits/sec
- - - - - - - - - - - - - - - - - - - - - - - - -
Test Complete. Summary Results:
[ ID][Role] Interval           Transfer     Bitrate         Retr
[  6][TX-C]   0.00-98.81  sec   500 MBytes  42.4 Mbits/sec  933             sender
[  6][TX-C]   0.00-98.82  sec   500 MBytes  42.4 Mbits/sec                  receiver
CPU Utilization: local/sender 3.5% (0.2%u/3.3%s), remote/receiver 16.6% (0.8%u/15.8%s)
CPU Utilization: local/receiver 3.5% (0.2%u/3.3%s), remote/sender 16.6% (0.8%u/15.8%s)
snd_tcp_congestion cubic
rcv_tcp_congestion cubic
[  8][RX-C]   0.00-98.81  sec   906 MBytes  76.9 Mbits/sec    0             sender
[  8][RX-C]   0.00-98.82  sec   905 MBytes  76.8 Mbits/sec                  receiver
snd_tcp_congestion cubic
rcv_tcp_congestion cubic

iperf Done.
```

A.1.6 TCP - 100 MB - Bidirectional

```
iperf 3.7
Linux tms 5.13.0-40-generic #45~20.04.1-Ubuntu SMP Mon Apr 4 09:38:31 UTC 2022 x86_64
Time: Sat, 26 Nov 2022 15:17:09 GMT
Connecting to host 192.168.8.20, port 5201
      Cookie: xwkoucuaca3qekx32ly2ry7zvlmi7e3fs24v
      TCP MSS: 1448 (default)
[  6] local 192.168.8.3 port 37570 connected to 192.168.8.20 port 5201
[  8] local 192.168.8.3 port 37574 connected to 192.168.8.20 port 5201
Starting Test: protocol: TCP, 1 streams, 131072 byte blocks, omitting 0 seconds,
104857600 bytes to send, tos 0
[ ID][Role] Interval           Transfer     Bitrate         Retr  Cwnd
[  6][TX-C]   0.00-1.00   sec  5.47 MBytes  45.9 Mbits/sec   14   41.0 KBytes
[  8][RX-C]   0.00-1.00   sec  8.62 MBytes  72.3 Mbits/sec
[  6][TX-C]   1.00-2.00   sec  6.24 MBytes  52.4 Mbits/sec   10   38.2 KBytes
[  8][RX-C]   1.00-2.00   sec  8.87 MBytes  74.4 Mbits/sec
[  6][TX-C]   2.00-3.00   sec  5.82 MBytes  48.8 Mbits/sec   17   33.9 KBytes
[  8][RX-C]   2.00-3.00   sec  8.97 MBytes  75.3 Mbits/sec
[  6][TX-C]   3.00-4.00   sec  6.05 MBytes  50.7 Mbits/sec    7   49.5 KBytes
[  8][RX-C]   3.00-4.00   sec  8.89 MBytes  74.6 Mbits/sec
[  6][TX-C]   4.00-5.00   sec  4.88 MBytes  40.9 Mbits/sec   11   52.3 KBytes
[  8][RX-C]   4.00-5.00   sec  9.24 MBytes  77.5 Mbits/sec
[  6][TX-C]   5.00-6.00   sec  5.65 MBytes  47.4 Mbits/sec   10   25.5 KBytes
[  8][RX-C]   5.00-6.00   sec  8.93 MBytes  74.9 Mbits/sec
[  6][TX-C]   6.00-7.00   sec  5.50 MBytes  46.1 Mbits/sec   12   36.8 KBytes
[  8][RX-C]   6.00-7.00   sec  9.04 MBytes  75.8 Mbits/sec
[  6][TX-C]   7.00-8.00   sec  5.42 MBytes  45.5 Mbits/sec   10   26.9 KBytes
[  8][RX-C]   7.00-8.00   sec  9.03 MBytes  75.7 Mbits/sec
[  6][TX-C]   8.00-9.00   sec  4.95 MBytes  41.5 Mbits/sec    5   48.1 KBytes
[  8][RX-C]   8.00-9.00   sec  9.21 MBytes  77.3 Mbits/sec
[  6][TX-C]   9.00-10.00  sec  5.15 MBytes  43.2 Mbits/sec    7   41.0 KBytes
[  8][RX-C]   9.00-10.00  sec  9.05 MBytes  75.9 Mbits/sec
[  6][TX-C]  10.00-11.00  sec  5.15 MBytes  43.2 Mbits/sec    8   43.8 KBytes
[  8][RX-C]  10.00-11.00  sec  9.16 MBytes  76.9 Mbits/sec
[  6][TX-C]  11.00-12.00  sec  5.35 MBytes  44.9 Mbits/sec    9   36.8 KBytes
[  8][RX-C]  11.00-12.00  sec  9.05 MBytes  75.9 Mbits/sec
[  6][TX-C]  12.00-13.00  sec  5.62 MBytes  47.1 Mbits/sec    7   45.2 KBytes
[  8][RX-C]  12.00-13.00  sec  9.05 MBytes  75.9 Mbits/sec
[  6][TX-C]  13.00-14.00  sec  5.54 MBytes  46.5 Mbits/sec   11   35.4 KBytes
[  8][RX-C]  13.00-14.00  sec  9.02 MBytes  75.6 Mbits/sec
[  6][TX-C]  14.00-15.00  sec  4.19 MBytes  35.2 Mbits/sec   10   43.8 KBytes
[  8][RX-C]  14.00-15.00  sec  9.42 MBytes  79.0 Mbits/sec
[  6][TX-C]  15.00-16.00  sec  5.53 MBytes  46.4 Mbits/sec    5   46.7 KBytes
[  8][RX-C]  15.00-16.00  sec  9.04 MBytes  75.8 Mbits/sec
[  6][TX-C]  16.00-17.00  sec  5.36 MBytes  45.0 Mbits/sec    6   59.4 KBytes
[  8][RX-C]  16.00-17.00  sec  9.02 MBytes  75.7 Mbits/sec
[  6][TX-C]  17.00-18.00  sec  5.51 MBytes  46.3 Mbits/sec   11   46.7 KBytes
[  8][RX-C]  17.00-18.00  sec  9.00 MBytes  75.5 Mbits/sec
[  6][TX-C]  18.00-18.51  sec  2.73 MBytes  44.9 Mbits/sec    6   53.7 KBytes
[  8][RX-C]  18.00-18.51  sec  4.66 MBytes  76.5 Mbits/sec
- - - - - - - - - - - - - - - - - - - - - - - - -
Test Complete. Summary Results:
[ ID][Role] Interval           Transfer     Bitrate         Retr
[  6][TX-C]   0.00-18.51  sec   100 MBytes  45.4 Mbits/sec  176             sender
[  6][TX-C]   0.00-18.51  sec  99.8 MBytes  45.2 Mbits/sec                  receiver
CPU Utilization: local/sender 3.7% (0.2%u/3.5%s), remote/receiver 7.9% (0.5%u/7.4%s)
CPU Utilization: local/receiver 3.7% (0.2%u/3.5%s), remote/sender 7.9% (0.5%u/7.4%s)
snd_tcp_congestion cubic
rcv_tcp_congestion cubic
[  8][RX-C]   0.00-18.51  sec   168 MBytes  76.0 Mbits/sec    0             sender
[  8][RX-C]   0.00-18.51  sec   167 MBytes  75.8 Mbits/sec                  receiver
snd_tcp_congestion cubic
rcv_tcp_congestion cubic

iperf Done.
```

A.1.7 TCP - 64 KB - Uplink

```
iperf 3.7
Linux tms 5.13.0-40-generic #45~20.04.1-Ubuntu SMP Mon Apr 4 09:38:31 UTC 2022 x86_64
Time: Sat, 26 Nov 2022 14:00:32 GMT
Connecting to host 192.168.8.20, port 5201
      Cookie: nqbyhliu7e4msarxg6tndjlus5pwgiqyuy4l
      TCP MSS: 1448 (default)
[  6] local 192.168.8.3 port 37444 connected to 192.168.8.20 port 5201
Starting Test: protocol: TCP, 1 streams, 64 byte blocks, omitting 0 seconds, 10 second test, tos 0
[ ID] Interval           Transfer     Bitrate         Retr  Cwnd
[  6]   0.00-1.00   sec  2.31 MBytes  19.4 Mbits/sec    1   19.8 KBytes
[  6]   1.00-2.00   sec  2.20 MBytes  18.4 Mbits/sec    2   15.6 KBytes
[  6]   2.00-3.00   sec  2.20 MBytes  18.4 Mbits/sec    0   19.8 KBytes
[  6]   3.00-4.00   sec  2.11 MBytes  17.7 Mbits/sec    2   21.2 KBytes
[  6]   4.00-5.00   sec  1.99 MBytes  16.7 Mbits/sec    3   21.2 KBytes
[  6]   5.00-6.00   sec  2.16 MBytes  18.1 Mbits/sec    2   22.6 KBytes
[  6]   6.00-7.00   sec  2.20 MBytes  18.4 Mbits/sec    1   19.8 KBytes
[  6]   7.00-8.00   sec  2.04 MBytes  17.1 Mbits/sec    0   19.8 KBytes
[  6]   8.00-9.00   sec  2.17 MBytes  18.2 Mbits/sec    1   19.8 KBytes
[  6]   9.00-10.00  sec  2.19 MBytes  18.3 Mbits/sec    1   21.2 KBytes
- - - - - - - - - - - - - - - - - - - - - - - - -
Test Complete. Summary Results:
[ ID] Interval           Transfer     Bitrate         Retr
[  6]   0.00-10.00  sec  21.6 MBytes  18.1 Mbits/sec   13             sender
[  6]   0.00-10.00  sec  21.4 MBytes  18.0 Mbits/sec                  receiver
CPU Utilization: local/sender 11.2% (2.6%u/8.6%s), remote/receiver 48.2% (3.6%u/44.6%s)
snd_tcp_congestion cubic
rcv_tcp_congestion cubic

iperf Done.
```

A.1.8 TCP - 100 KB - Uplink

```
iperf 3.7
Linux tms 5.13.0-40-generic #45~20.04.1-Ubuntu SMP Mon Apr 4 09:38:31 UTC 2022 x86_64
Time: Sat, 26 Nov 2022 14:00:12 GMT
Connecting to host 192.168.8.20, port 5201
      Cookie: 5qcyowny4wwsclvdqmncpaq34m52l7iwt77a
      TCP MSS: 1448 (default)
[  6] local 192.168.8.3 port 37440 connected to 192.168.8.20 port 5201
Starting Test: protocol: TCP, 1 streams, 100 byte blocks, omitting 0 seconds, 10 second test, tos 0
[ ID] Interval           Transfer     Bitrate         Retr  Cwnd
[  6]   0.00-1.00   sec  3.51 MBytes  29.4 Mbits/sec    4   21.2 KBytes
[  6]   1.00-2.00   sec  3.34 MBytes  28.0 Mbits/sec    2   19.8 KBytes
[  6]   2.00-3.00   sec  3.33 MBytes  28.0 Mbits/sec    2   19.8 KBytes
[  6]   3.00-4.00   sec  3.33 MBytes  28.0 Mbits/sec    4   21.2 KBytes
[  6]   4.00-5.00   sec  3.34 MBytes  28.0 Mbits/sec    0   21.2 KBytes
[  6]   5.00-6.00   sec  3.35 MBytes  28.1 Mbits/sec    2   21.2 KBytes
[  6]   6.00-7.00   sec  3.34 MBytes  28.0 Mbits/sec    3   22.6 KBytes
[  6]   7.00-8.00   sec  3.29 MBytes  27.6 Mbits/sec    0   22.6 KBytes
[  6]   8.00-9.00   sec  3.10 MBytes  26.0 Mbits/sec    2   19.8 KBytes
[  6]   9.00-10.00  sec  3.32 MBytes  27.9 Mbits/sec    1   21.2 KBytes
- - - - - - - - - - - - - - - - - - - - - - - - -
Test Complete. Summary Results:
[ ID] Interval           Transfer     Bitrate         Retr
[  6]   0.00-10.00  sec  33.3 MBytes  27.9 Mbits/sec   20             sender
[  6]   0.00-10.00  sec  33.1 MBytes  27.7 Mbits/sec                  receiver
CPU Utilization: local/sender 13.8% (2.3%u/11.5%s), remote/receiver 55.4% (3.8%u/51.6%s)
snd_tcp_congestion cubic
rcv_tcp_congestion cubic

iperf Done.
```

A.1.9 TCP - 500 KB - Uplink

```
iperf 3.7
Linux tms 5.13.0-40-generic #45~20.04.1-Ubuntu SMP Mon Apr 4 09:38:31 UTC 2022 x86_64
Time: Sat, 26 Nov 2022 13:59:54 GMT
Connecting to host 192.168.8.20, port 5201
      Cookie: kgqysapsjwaaboabhanhtxezqfvmgfi3cu7d
      TCP MSS: 1448 (default)
[  6] local 192.168.8.3 port 37436 connected to 192.168.8.20 port 5201
Starting Test: protocol: TCP, 1 streams, 500 byte blocks, omitting 0 seconds, 10 second test, tos 0
[ ID] Interval           Transfer     Bitrate         Retr  Cwnd
[  6]   0.00-1.00   sec  11.3 MBytes  95.0 Mbits/sec    3    100 KBytes
[  6]   1.00-2.00   sec  11.2 MBytes  94.2 Mbits/sec    6   59.4 KBytes
[  6]   2.00-3.00   sec  10.7 MBytes  89.4 Mbits/sec   17   66.5 KBytes
[  6]   3.00-4.00   sec  11.1 MBytes  93.5 Mbits/sec    4   87.7 KBytes
[  6]   4.00-5.00   sec  11.0 MBytes  92.5 Mbits/sec    1    124 KBytes
[  6]   5.00-6.00   sec  10.7 MBytes  89.6 Mbits/sec   49   73.5 KBytes
[  6]   6.00-7.00   sec  10.3 MBytes  86.7 Mbits/sec    8   32.5 KBytes
[  6]   7.00-8.00   sec  9.81 MBytes  82.3 Mbits/sec    6   32.5 KBytes
[  6]   8.00-9.00   sec  11.0 MBytes  92.6 Mbits/sec    3    102 KBytes
[  6]   9.00-10.00  sec  10.9 MBytes  91.5 Mbits/sec    9   49.5 KBytes
- - - - - - - - - - - - - - - - - - - - - - - - -
Test Complete. Summary Results:
[ ID] Interval           Transfer     Bitrate         Retr
[  6]   0.00-10.00  sec   108 MBytes  90.7 Mbits/sec  106             sender
[  6]   0.00-10.00  sec   108 MBytes  90.3 Mbits/sec                  receiver
CPU Utilization: local/sender 4.7% (0.8%u/4.0%s), remote/receiver 22.8% (1.3%u/21.6%s)
snd_tcp_congestion cubic
rcv_tcp_congestion cubic

iperf Done.
```

A.1.10 TCP - 1448 KB - Uplink

```
iperf 3.7
Linux tms 5.13.0-40-generic #45~20.04.1-Ubuntu SMP Mon Apr 4 09:38:31 UTC 2022 x86_64
Time: Sat, 26 Nov 2022 13:59:14 GMT
Connecting to host 192.168.8.20, port 5201
      Cookie: modf6humwyh62siocsl37edqaysb2loqe724
      TCP MSS: 1448 (default)
[  6] local 192.168.8.3 port 37432 connected to 192.168.8.20 port 5201
Starting Test: protocol: TCP, 1 streams, 1448 byte blocks, omitting 0 seconds, 10 second test, tos 0
[ ID] Interval           Transfer     Bitrate         Retr  Cwnd
[  6]   0.00-1.00   sec  11.5 MBytes  96.8 Mbits/sec    6   49.5 KBytes
[  6]   1.00-2.00   sec  11.1 MBytes  92.9 Mbits/sec   26   43.8 KBytes
[  6]   2.00-3.00   sec  11.2 MBytes  94.1 Mbits/sec    5   55.1 KBytes
[  6]   3.00-4.00   sec  11.2 MBytes  94.3 Mbits/sec    5   62.2 KBytes
[  6]   4.00-5.00   sec  11.1 MBytes  92.8 Mbits/sec   27   39.6 KBytes
[  6]   5.00-6.00   sec  11.2 MBytes  94.1 Mbits/sec    3   96.2 KBytes
[  6]   6.00-7.00   sec  11.2 MBytes  94.2 Mbits/sec    5   84.8 KBytes
[  6]   7.00-8.00   sec  11.1 MBytes  92.8 Mbits/sec   24   43.8 KBytes
[  6]   8.00-9.00   sec  11.2 MBytes  94.1 Mbits/sec    8   36.8 KBytes
[  6]   9.00-10.00  sec  11.2 MBytes  94.1 Mbits/sec    6   65.0 KBytes
- - - - - - - - - - - - - - - - - - - - - - - - -
Test Complete. Summary Results:
[ ID] Interval           Transfer     Bitrate         Retr
[  6]   0.00-10.00  sec   112 MBytes  94.0 Mbits/sec  115             sender
[  6]   0.00-10.01  sec   112 MBytes  93.7 Mbits/sec                  receiver
CPU Utilization: local/sender 4.1% (0.6%u/3.4%s), remote/receiver 2.7% (0.1%u/2.6%s)
snd_tcp_congestion cubic
rcv_tcp_congestion cubic

iperf Done.
```

A.1.11 TCP - 64 KB - Downlink

```
iperf 3.1.3
Linux ts-146 3.10.0-693.el7.x86_64 #1 SMP Tue Aug 22 21:09:27 UTC 2017 x86_64
Time: Sat, 26 Nov 2022 14:13:21 GMT
Connecting to host 192.168.8.20, port 5201
Reverse mode, remote host 192.168.8.20 is sending
      Cookie: ts-146.1669472001.172061.0522d46b7d0
      TCP MSS: 1448 (default)
[  4] local 192.168.8.5 port 40078 connected to 192.168.8.20 port 5201
Starting Test: protocol: TCP, 1 streams, 64 byte blocks, omitting 0 seconds, 10 second test
[ ID] Interval           Transfer     Bandwidth
[  4]   0.00-1.00   sec  4.16 MBytes  34.9 Mbits/sec
[  4]   1.00-2.00   sec  3.81 MBytes  32.0 Mbits/sec
[  4]   2.00-3.00   sec  4.01 MBytes  33.7 Mbits/sec
[  4]   3.00-4.00   sec  4.09 MBytes  34.3 Mbits/sec
[  4]   4.00-5.00   sec  3.69 MBytes  31.0 Mbits/sec
[  4]   5.00-6.00   sec  3.69 MBytes  30.9 Mbits/sec
[  4]   6.00-7.00   sec  3.69 MBytes  31.0 Mbits/sec
[  4]   7.00-8.00   sec  3.70 MBytes  31.1 Mbits/sec
[  4]   8.00-9.00   sec  3.69 MBytes  31.0 Mbits/sec
[  4]   9.00-10.00  sec  3.71 MBytes  31.1 Mbits/sec
- - - - - - - - - - - - - - - - - - - - - - - - -
Test Complete. Summary Results:
[ ID] Interval           Transfer     Bandwidth       Retr
[  4]   0.00-10.00  sec  38.3 MBytes  32.1 Mbits/sec    0             sender
[  4]   0.00-10.00  sec  38.3 MBytes  32.1 Mbits/sec                  receiver
CPU Utilization: local/receiver 28.2% (3.7%u/24.5%s), remote/sender 3.8% (0.2%u/3.6%s)

iperf Done.
```

A.1.12 TCP - 100 KB - Downlink

```
iperf 3.1.3
Linux ts-146 3.10.0-693.el7.x86_64 #1 SMP Tue Aug 22 21:09:27 UTC 2017 x86_64
Time: Sat, 26 Nov 2022 14:07:05 GMT
Connecting to host 192.168.8.20, port 5201
Reverse mode, remote host 192.168.8.20 is sending
      Cookie: ts-146.1669471625.574105.5bae26e5601
      TCP MSS: 1448 (default)
[  4] local 192.168.8.5 port 40070 connected to 192.168.8.20 port 5201
Starting Test: protocol: TCP, 1 streams, 100 byte blocks, omitting 0 seconds, 10 second test
[ ID] Interval           Transfer     Bandwidth
[  4]   0.00-1.00   sec  6.32 MBytes  53.0 Mbits/sec
[  4]   1.00-2.00   sec  6.29 MBytes  52.7 Mbits/sec
[  4]   2.00-3.00   sec  6.27 MBytes  52.6 Mbits/sec
[  4]   3.00-4.00   sec  6.29 MBytes  52.7 Mbits/sec
[  4]   4.00-5.00   sec  6.14 MBytes  51.5 Mbits/sec
[  4]   5.00-6.00   sec  6.34 MBytes  53.1 Mbits/sec
[  4]   6.00-7.00   sec  6.26 MBytes  52.5 Mbits/sec
[  4]   7.00-8.00   sec  6.32 MBytes  53.0 Mbits/sec
[  4]   8.00-9.00   sec  6.32 MBytes  53.0 Mbits/sec
[  4]   9.00-10.00  sec  6.32 MBytes  53.0 Mbits/sec
- - - - - - - - - - - - - - - - - - - - - - - - -
Test Complete. Summary Results:
[ ID] Interval           Transfer     Bandwidth       Retr
[  4]   0.00-10.00  sec  62.9 MBytes  52.7 Mbits/sec    0             sender
[  4]   0.00-10.00  sec  62.9 MBytes  52.7 Mbits/sec                  receiver
CPU Utilization: local/receiver 32.3% (4.2%u/28.1%s), remote/sender 33.0% (1.3%u/31.7%s)

iperf Done.
```

A.1.13 TCP - 500 KB - Downlink

```
iperf 3.7
Linux tms 5.13.0-40-generic #45~20.04.1-Ubuntu SMP Mon Apr 4 09:38:31 UTC 2022 x86_64
Time: Sat, 26 Nov 2022 14:03:49 GMT
Connecting to host 192.168.8.20, port 5201
Reverse mode, remote host 192.168.8.20 is sending
      Cookie: 4kz5mmthla3sfbw3g73j3gv4g3huajtjj6i5
      TCP MSS: 1448 (default)
[  6] local 192.168.8.3 port 37452 connected to 192.168.8.20 port 5201
Starting Test: protocol: TCP, 1 streams, 500 byte blocks, omitting 0 seconds, 10 second test, tos 0
[ ID] Interval           Transfer     Bitrate
[  6]   0.00-1.00   sec  10.8 MBytes  90.2 Mbits/sec
[  6]   1.00-2.00   sec  10.8 MBytes  90.3 Mbits/sec
[  6]   2.00-3.00   sec  10.8 MBytes  91.0 Mbits/sec
[  6]   3.00-4.00   sec  10.8 MBytes  90.2 Mbits/sec
[  6]   4.00-5.00   sec  10.9 MBytes  91.8 Mbits/sec
[  6]   5.00-6.00   sec  10.8 MBytes  90.8 Mbits/sec
[  6]   6.00-7.00   sec  10.8 MBytes  90.8 Mbits/sec
[  6]   7.00-8.00   sec  10.8 MBytes  90.8 Mbits/sec
[  6]   8.00-9.00   sec  10.8 MBytes  90.6 Mbits/sec
[  6]   9.00-10.00  sec  10.8 MBytes  90.9 Mbits/sec
- - - - - - - - - - - - - - - - - - - - - - - - -
Test Complete. Summary Results:
[ ID] Interval           Transfer     Bitrate         Retr
[  6]   0.00-10.00  sec   109 MBytes  91.0 Mbits/sec    0             sender
[  6]   0.00-10.00  sec   108 MBytes  90.8 Mbits/sec                  receiver
snd_tcp_congestion cubic
rcv_tcp_congestion cubic

iperf Done.
```

A.1.14 TCP - 1448 KB - Downlink

```
iperf 3.7
Linux tms 5.13.0-40-generic #45~20.04.1-Ubuntu SMP Mon Apr 4 09:38:31 UTC 2022 x86_64
Time: Sat, 26 Nov 2022 14:03:24 GMT
Connecting to host 192.168.8.20, port 5201
Reverse mode, remote host 192.168.8.20 is sending
      Cookie: i5hpil7gkwg52p4dughc66f7jlduedhsrsk4
      TCP MSS: 1448 (default)
[  6] local 192.168.8.3 port 37448 connected to 192.168.8.20 port 5201
Starting Test: protocol: TCP, 1 streams, 1448 byte blocks, omitting 0 seconds, 10 second test, tos 0
[ ID] Interval           Transfer     Bitrate
[  6]   0.00-1.00   sec  10.8 MBytes  90.7 Mbits/sec
[  6]   1.00-2.00   sec  10.9 MBytes  91.4 Mbits/sec
[  6]   2.00-3.00   sec  10.9 MBytes  91.1 Mbits/sec
[  6]   3.00-4.00   sec  10.9 MBytes  91.4 Mbits/sec
[  6]   4.00-5.00   sec  10.9 MBytes  91.2 Mbits/sec
[  6]   5.00-6.00   sec  10.8 MBytes  90.6 Mbits/sec
[  6]   6.00-7.00   sec  10.8 MBytes  90.2 Mbits/sec
[  6]   7.00-8.00   sec  10.8 MBytes  90.6 Mbits/sec
[  6]   8.00-9.00   sec  10.8 MBytes  90.3 Mbits/sec
[  6]   9.00-10.00  sec  10.9 MBytes  91.1 Mbits/sec
- - - - - - - - - - - - - - - - - - - - - - - - -
Test Complete. Summary Results:
[ ID] Interval           Transfer     Bitrate         Retr
[  6]   0.00-10.00  sec   109 MBytes  91.1 Mbits/sec    0             sender
[  6]   0.00-10.00  sec   108 MBytes  90.9 Mbits/sec                  receiver
snd_tcp_congestion cubic
rcv_tcp_congestion cubic

iperf Done.
```

A.1.15 TCP - 64 KB - Bidirectional

```
iperf 3.7
Linux tms 5.13.0-40-generic #45~20.04.1-Ubuntu SMP Mon Apr 4 09:38:31 UTC 2022 x86_64
Time: Sat, 26 Nov 2022 14:46:34 GMT
Connecting to host 192.168.8.20, port 5201
      Cookie: btdxofn3wmjuaoxslp3gehfi7fmaw2flql6v
      TCP MSS: 1448 (default)
[  6] local 192.168.8.3 port 37490 connected to 192.168.8.20 port 5201
[  8] local 192.168.8.3 port 37492 connected to 192.168.8.20 port 5201
Starting Test: protocol: TCP, 1 streams, 64 byte blocks, omitting 0 seconds, 10 second test, tos 0
[ ID][Role] Interval           Transfer     Bitrate         Retr  Cwnd
[  6][TX-C]   0.00-1.00   sec   547 KBytes  4.48 Mbits/sec    0   32.5 KBytes
[  8][RX-C]   0.00-1.00   sec  3.45 MBytes  29.0 Mbits/sec
[  6][TX-C]   1.00-2.00   sec   423 KBytes  3.47 Mbits/sec    0   32.5 KBytes
[  8][RX-C]   1.00-2.00   sec  4.07 MBytes  34.1 Mbits/sec
[  6][TX-C]   2.00-3.00   sec   404 KBytes  3.31 Mbits/sec    1   22.6 KBytes
[  8][RX-C]   2.00-3.00   sec  4.09 MBytes  34.3 Mbits/sec
[  6][TX-C]   3.00-4.00   sec   422 KBytes  3.46 Mbits/sec    1   17.0 KBytes
[  8][RX-C]   3.00-4.00   sec  4.09 MBytes  34.3 Mbits/sec
[  6][TX-C]   4.00-5.00   sec   402 KBytes  3.29 Mbits/sec    1   15.6 KBytes
[  8][RX-C]   4.00-5.00   sec  3.94 MBytes  33.0 Mbits/sec
[  6][TX-C]   5.00-6.00   sec   436 KBytes  3.57 Mbits/sec    3   8.48 KBytes
[  8][RX-C]   5.00-6.00   sec  4.06 MBytes  34.1 Mbits/sec
[  6][TX-C]   6.00-7.00   sec   405 KBytes  3.32 Mbits/sec    1   19.8 KBytes
[  8][RX-C]   6.00-7.00   sec  4.07 MBytes  34.2 Mbits/sec
[  6][TX-C]   7.00-8.00   sec   423 KBytes  3.47 Mbits/sec    3   19.8 KBytes
[  8][RX-C]   7.00-8.00   sec  4.09 MBytes  34.3 Mbits/sec
[  6][TX-C]   8.00-9.00   sec   408 KBytes  3.34 Mbits/sec    1   18.4 KBytes
[  8][RX-C]   8.00-9.00   sec  4.08 MBytes  34.2 Mbits/sec
[  6][TX-C]   9.00-10.00  sec   431 KBytes  3.53 Mbits/sec    3   21.2 KBytes
[  8][RX-C]   9.00-10.00  sec  4.08 MBytes  34.2 Mbits/sec
- - - - - - - - - - - - - - - - - - - - - - - - -
```

```
Test Complete. Summary Results:
[ ID][Role] Interval           Transfer     Bitrate         Retr
[  6][TX-C]   0.00-10.00  sec  4.20 MBytes  3.52 Mbits/sec   14             sender
[  6][TX-C]   0.00-10.00  sec  4.00 MBytes  3.36 Mbits/sec                  receiver
CPU Utilization: local/sender 13.0% (2.4%u/10.6%s), remote/receiver 11.9% (0.7%u/11.1%s)
CPU Utilization: local/receiver 13.0% (2.4%u/10.6%s), remote/sender 11.9% (0.7%u/11.1%s)
snd_tcp_congestion cubic
rcv_tcp_congestion cubic
[  8][RX-C]   0.00-10.00  sec  40.0 MBytes  33.6 Mbits/sec    0             sender
[  8][RX-C]   0.00-10.00  sec  40.0 MBytes  33.6 Mbits/sec                  receiver
snd_tcp_congestion cubic
rcv_tcp_congestion cubic

iperf Done.
```

A.1.16 TCP - 100 KB - Bidirectional

```
iperf 3.7
Linux tms 5.13.0-40-generic #45~20.04.1-Ubuntu SMP Mon Apr 4 09:38:31 UTC 2022 x86_64
Time: Sat, 26 Nov 2022 14:45:11 GMT
Connecting to host 192.168.8.20, port 5201
      Cookie: bd4b3knwisig6dnexldnsuu76l54szualcxf
      TCP MSS: 1448 (default)
[  6] local 192.168.8.3 port 37484 connected to 192.168.8.20 port 5201
[  8] local 192.168.8.3 port 37486 connected to 192.168.8.20 port 5201
Starting Test: protocol: TCP, 1 streams, 100 byte blocks, omitting 0 seconds, 10 second test, tos 0
[ ID][Role] Interval           Transfer     Bitrate         Retr  Cwnd
[  6][TX-C]   0.00-1.00   sec   679 KBytes  5.56 Mbits/sec    3   15.6 KBytes
[  8][RX-C]   0.00-1.00   sec  4.51 MBytes  37.9 Mbits/sec
[  6][TX-C]   1.00-2.00   sec   455 KBytes  3.73 Mbits/sec    0   21.2 KBytes
[  8][RX-C]   1.00-2.00   sec  4.53 MBytes  38.0 Mbits/sec
[  6][TX-C]   2.00-3.00   sec   469 KBytes  3.84 Mbits/sec    0   21.2 KBytes
[  8][RX-C]   2.00-3.00   sec  4.54 MBytes  38.1 Mbits/sec
[  6][TX-C]   3.00-4.00   sec   472 KBytes  3.86 Mbits/sec    1   19.8 KBytes
[  8][RX-C]   3.00-4.00   sec  4.55 MBytes  38.2 Mbits/sec
[  6][TX-C]   4.00-5.00   sec   445 KBytes  3.65 Mbits/sec    2   21.2 KBytes
[  8][RX-C]   4.00-5.00   sec  4.54 MBytes  38.1 Mbits/sec
[  6][TX-C]   5.00-6.00   sec   470 KBytes  3.85 Mbits/sec    0   21.2 KBytes
[  8][RX-C]   5.00-6.00   sec  4.56 MBytes  38.2 Mbits/sec
[  6][TX-C]   6.00-7.00   sec   476 KBytes  3.90 Mbits/sec    1   19.8 KBytes
[  8][RX-C]   6.00-7.00   sec  4.55 MBytes  38.1 Mbits/sec
[  6][TX-C]   7.00-8.00   sec   453 KBytes  3.71 Mbits/sec    1   21.2 KBytes
[  8][RX-C]   7.00-8.00   sec  4.55 MBytes  38.2 Mbits/sec
[  6][TX-C]   8.00-9.00   sec   474 KBytes  3.88 Mbits/sec    3   11.3 KBytes
[  8][RX-C]   8.00-9.00   sec  4.51 MBytes  37.8 Mbits/sec
[  6][TX-C]   9.00-10.00  sec   476 KBytes  3.90 Mbits/sec    1   12.7 KBytes
[  8][RX-C]   9.00-10.00  sec  4.55 MBytes  38.1 Mbits/sec
- - - - - - - - - - - - - - - - - - - - - - - - -
Test Complete. Summary Results:
[ ID][Role] Interval           Transfer     Bitrate         Retr
[  6][TX-C]   0.00-10.00  sec  4.75 MBytes  3.99 Mbits/sec   12             sender
[  6][TX-C]   0.00-10.00  sec  4.54 MBytes  3.81 Mbits/sec                  receiver
CPU Utilization: local/sender 10.6% (1.8%u/8.8%s), remote/receiver 1.4% (0.1%u/1.3%s)
CPU Utilization: local/receiver 10.6% (1.8%u/8.8%s), remote/sender 1.4% (0.1%u/1.3%s)
snd_tcp_congestion cubic
rcv_tcp_congestion cubic
[  8][RX-C]   0.00-10.00  sec  45.4 MBytes  38.1 Mbits/sec    0             sender
[  8][RX-C]   0.00-10.00  sec  45.4 MBytes  38.1 Mbits/sec                  receiver
snd_tcp_congestion cubic
rcv_tcp_congestion cubic

iperf Done.
```

A.1.17 TCP - 500 KB - Bidirectional

```
iperf 3.7
Linux tms 5.13.0-40-generic #45~20.04.1-Ubuntu SMP Mon Apr 4 09:38:31 UTC 2022 x86_64
Time: Sat, 26 Nov 2022 14:20:09 GMT
Connecting to host 192.168.8.20, port 5201
      Cookie: hy7bytfwbfgqsgyoo5ybfnbbco6nlrj3jpkh
      TCP MSS: 1448 (default)
[  6] local 192.168.8.3 port 37472 connected to 192.168.8.20 port 5201
[  8] local 192.168.8.3 port 37474 connected to 192.168.8.20 port 5201
Starting Test: protocol: TCP, 1 streams, 500 byte blocks, omitting 0 seconds, 10 second test, tos 0
[ ID][Role] Interval           Transfer     Bitrate         Retr  Cwnd
[  6][TX-C]   0.00-1.00   sec  6.09 MBytes  51.1 Mbits/sec   11   38.2 KBytes
[  8][RX-C]   0.00-1.00   sec  8.57 MBytes  71.9 Mbits/sec
[  6][TX-C]   1.00-2.00   sec  5.47 MBytes  45.9 Mbits/sec    8   32.5 KBytes
[  8][RX-C]   1.00-2.00   sec  8.86 MBytes  74.3 Mbits/sec
[  6][TX-C]   2.00-3.00   sec  5.28 MBytes  44.3 Mbits/sec    8   19.8 KBytes
[  8][RX-C]   2.00-3.00   sec  8.84 MBytes  74.1 Mbits/sec
[  6][TX-C]   3.00-4.00   sec  4.42 MBytes  37.0 Mbits/sec    7   31.1 KBytes
[  8][RX-C]   3.00-4.00   sec  9.21 MBytes  77.3 Mbits/sec
[  6][TX-C]   4.00-5.00   sec  4.79 MBytes  40.2 Mbits/sec   10   46.7 KBytes
[  8][RX-C]   4.00-5.00   sec  9.11 MBytes  76.4 Mbits/sec
[  6][TX-C]   5.00-6.00   sec  4.81 MBytes  40.3 Mbits/sec    7   50.9 KBytes
[  8][RX-C]   5.00-6.00   sec  9.19 MBytes  77.1 Mbits/sec
[  6][TX-C]   6.00-7.00   sec  4.97 MBytes  41.7 Mbits/sec    4   35.4 KBytes
[  8][RX-C]   6.00-7.00   sec  9.13 MBytes  76.6 Mbits/sec
[  6][TX-C]   7.00-8.00   sec  5.35 MBytes  44.8 Mbits/sec   10   31.1 KBytes
[  8][RX-C]   7.00-8.00   sec  8.89 MBytes  74.6 Mbits/sec
[  6][TX-C]   8.00-9.00   sec  5.35 MBytes  44.8 Mbits/sec    8   21.2 KBytes
[  8][RX-C]   8.00-9.00   sec  8.95 MBytes  75.1 Mbits/sec
[  6][TX-C]   9.00-10.00  sec  4.79 MBytes  40.2 Mbits/sec    7   28.3 KBytes
[  8][RX-C]   9.00-10.00  sec  9.09 MBytes  76.3 Mbits/sec
- - - - - - - - - - - - - - - - - - - - - - - - -
Test Complete. Summary Results:
[ ID][Role] Interval           Transfer     Bitrate         Retr
[  6][TX-C]   0.00-10.00  sec  51.3 MBytes  43.0 Mbits/sec   80             sender
[  6][TX-C]   0.00-10.00  sec  51.1 MBytes  42.9 Mbits/sec                  receiver
CPU Utilization: local/sender 4.7% (0.8%u/3.9%s), remote/receiver 9.0% (0.6%u/8.4%s)
CPU Utilization: local/receiver 4.7% (0.8%u/3.9%s), remote/sender 9.0% (0.6%u/8.4%s)
snd_tcp_congestion cubic
rcv_tcp_congestion cubic
[  8][RX-C]   0.00-10.00  sec  90.1 MBytes  75.6 Mbits/sec    0             sender
[  8][RX-C]   0.00-10.00  sec  89.8 MBytes  75.4 Mbits/sec                  receiver
snd_tcp_congestion cubic
rcv_tcp_congestion cubic

iperf Done.
```

A.1.18 TCP - 1448 KB - Bidirectional

```
iperf 3.7
Linux tms 5.13.0-40-generic #45~20.04.1-Ubuntu SMP Mon Apr 4 09:38:31 UTC 2022 x86_64
Time: Sat, 26 Nov 2022 14:18:50 GMT
Connecting to host 192.168.8.20, port 5201
      Cookie: edothnq52va6kqnggyzgejma43hs2nobbmll
      TCP MSS: 1448 (default)
[  6] local 192.168.8.3 port 37466 connected to 192.168.8.20 port 5201
[  8] local 192.168.8.3 port 37468 connected to 192.168.8.20 port 5201
Starting Test: protocol: TCP, 1 streams, 1448 byte blocks, omitting 0 seconds, 10 second test, tos 0
[ ID][Role] Interval           Transfer     Bitrate         Retr  Cwnd
[  6][TX-C]   0.00-1.00   sec  5.99 MBytes  50.3 Mbits/sec    8   33.9 KBytes
[  8][RX-C]   0.00-1.00   sec  8.69 MBytes  72.9 Mbits/sec
[  6][TX-C]   1.00-2.00   sec  5.65 MBytes  47.4 Mbits/sec   10   52.3 KBytes
[  8][RX-C]   1.00-2.00   sec  8.76 MBytes  73.5 Mbits/sec
[  6][TX-C]   2.00-3.00   sec  6.32 MBytes  53.1 Mbits/sec   12   26.9 KBytes
[  8][RX-C]   2.00-3.00   sec  8.63 MBytes  72.4 Mbits/sec
[  6][TX-C]   3.00-4.00   sec  5.65 MBytes  47.4 Mbits/sec    5   45.2 KBytes
[  8][RX-C]   3.00-4.00   sec  8.85 MBytes  74.3 Mbits/sec
[  6][TX-C]   4.00-5.00   sec  5.26 MBytes  44.1 Mbits/sec   11   26.9 KBytes
[  8][RX-C]   4.00-5.00   sec  8.91 MBytes  74.7 Mbits/sec
[  6][TX-C]   5.00-6.00   sec  5.43 MBytes  45.5 Mbits/sec    7   33.9 KBytes
[  8][RX-C]   5.00-6.00   sec  8.84 MBytes  74.1 Mbits/sec
[  6][TX-C]   6.00-7.00   sec  5.70 MBytes  47.8 Mbits/sec    5   53.7 KBytes
[  8][RX-C]   6.00-7.00   sec  8.81 MBytes  73.9 Mbits/sec
[  6][TX-C]   7.00-8.00   sec  5.37 MBytes  45.1 Mbits/sec    8   24.0 KBytes
[  8][RX-C]   7.00-8.00   sec  8.93 MBytes  74.9 Mbits/sec
[  6][TX-C]   8.00-9.00   sec  4.75 MBytes  39.9 Mbits/sec    6   33.9 KBytes
[  8][RX-C]   8.00-9.00   sec  9.08 MBytes  76.1 Mbits/sec
[  6][TX-C]   9.00-10.00  sec  5.94 MBytes  49.8 Mbits/sec    9   39.6 KBytes
[  8][RX-C]   9.00-10.00  sec  8.77 MBytes  73.5 Mbits/sec
- - - - - - - - - - - - - - - - - - - - - - - - -
Test Complete. Summary Results:
[ ID][Role] Interval           Transfer     Bitrate         Retr
[  6][TX-C]   0.00-10.00  sec  56.1 MBytes  47.0 Mbits/sec   81             sender
[  6][TX-C]   0.00-10.00  sec  55.9 MBytes  46.9 Mbits/sec                  receiver
CPU Utilization: local/sender 3.1% (0.5%u/2.7%s), remote/receiver 20.3% (1.1%u/19.3%s)
CPU Utilization: local/receiver 3.1% (0.5%u/2.7%s), remote/sender 20.3% (1.1%u/19.3%s)
snd_tcp_congestion cubic
rcv_tcp_congestion cubic
[  8][RX-C]   0.00-10.00  sec  88.5 MBytes  74.3 Mbits/sec    0             sender
[  8][RX-C]   0.00-10.00  sec  88.3 MBytes  74.0 Mbits/sec                  receiver
snd_tcp_congestion cubic
rcv_tcp_congestion cubic

iperf Done.
```

A.2 UDP

A.2.1 UDP - 64 KB - 4.1 Mbps - Uplink

```
iperf 3.7
Linux tms 5.13.0-40-generic #45~20.04.1-Ubuntu SMP Mon Apr 4 09:38:31 UTC 2022 x86_64
Time: Sat, 26 Nov 2022 14:59:56 GMT
Connecting to host 192.168.8.20, port 5201
      Cookie: 6rhxrewedvet3rtirivyl4hndyokj5qndaad
      Target Bitrate: 4099999
[  6] local 192.168.8.3 port 58344 connected to 192.168.8.20 port 5201
Starting Test: protocol: UDP, 1 streams, 64 byte blocks, omitting 0 seconds, 10 second test, tos 0
[ ID] Interval           Transfer     Bitrate         Total Datagrams
[  6]   0.00-1.00   sec   500 KBytes  4.10 Mbits/sec  8002
[  6]   1.00-2.00   sec   500 KBytes  4.10 Mbits/sec  8007
[  6]   2.00-3.00   sec   500 KBytes  4.10 Mbits/sec  8008
[  6]   3.00-4.00   sec   500 KBytes  4.10 Mbits/sec  8008
[  6]   4.00-5.00   sec   500 KBytes  4.10 Mbits/sec  8008
[  6]   5.00-6.00   sec   500 KBytes  4.10 Mbits/sec  8008
[  6]   6.00-7.00   sec   500 KBytes  4.10 Mbits/sec  8008
[  6]   7.00-8.00   sec   500 KBytes  4.10 Mbits/sec  8007
[  6]   8.00-9.00   sec   500 KBytes  4.10 Mbits/sec  8008
[  6]   9.00-10.00  sec   500 KBytes  4.10 Mbits/sec  8008
- - - - - - - - - - - - - - - - - - - - - - - - -
Test Complete. Summary Results:
[ ID] Interval           Transfer     Bitrate         Jitter    Lost/Total Datagrams
[  6]   0.00-10.00  sec  4.89 MBytes  4.10 Mbits/sec  0.000 ms  0/80072 (0%)  sender
[  6]   0.00-10.00  sec  4.88 MBytes  4.09 Mbits/sec  0.152 ms  93/80072 (0.12%)  receiver
CPU Utilization: local/sender 5.9% (0.0%u/5.9%s), remote/receiver 13.8% (0.8%u/13.0%s)

iperf Done.
```

A.2.2 UDP - 100 KB - 6.4 Mbps - Uplink

```
iperf 3.7
Linux tms 5.13.0-40-generic #45~20.04.1-Ubuntu SMP Mon Apr 4 09:38:31 UTC 2022 x86_64
Time: Sat, 26 Nov 2022 14:57:48 GMT
Connecting to host 192.168.8.20, port 5201
      Cookie: 62nt4lsbw3ja5qtc5fjpw2xtpbvd2ghufmtr
      Target Bitrate: 6400000
[  6] local 192.168.8.3 port 46333 connected to 192.168.8.20 port 5201
Starting Test: protocol: UDP, 1 streams, 100 byte blocks, omitting 0 seconds, 10 second test, tos 0
[ ID] Interval           Transfer     Bitrate         Total Datagrams
[  6]   0.00-1.00   sec   781 KBytes  6.40 Mbits/sec  7996
[  6]   1.00-2.00   sec   781 KBytes  6.40 Mbits/sec  7998
[  6]   2.00-3.00   sec   781 KBytes  6.40 Mbits/sec  8000
[  6]   3.00-4.00   sec   781 KBytes  6.40 Mbits/sec  7999
[  6]   4.00-5.00   sec   781 KBytes  6.40 Mbits/sec  8001
[  6]   5.00-6.00   sec   781 KBytes  6.40 Mbits/sec  8001
[  6]   6.00-7.00   sec   781 KBytes  6.40 Mbits/sec  8000
[  6]   7.00-8.00   sec   781 KBytes  6.40 Mbits/sec  8000
[  6]   8.00-9.00   sec   781 KBytes  6.40 Mbits/sec  8000
[  6]   9.00-10.00  sec   781 KBytes  6.40 Mbits/sec  8000
- - - - - - - - - - - - - - - - - - - - - - - - -
Test Complete. Summary Results:
[ ID] Interval           Transfer     Bitrate         Jitter    Lost/Total Datagrams
[  6]   0.00-10.00  sec  7.63 MBytes  6.40 Mbits/sec  0.000 ms  0/79995 (0%)  sender
[  6]   0.00-10.00  sec  7.62 MBytes  6.39 Mbits/sec  0.183 ms  117/79995 (0.15%)  receiver
CPU Utilization: local/sender 13.6% (13.6%u/0.0%s), remote/receiver 25.1% (1.6%u/23.5%s)

iperf Done.
```

A.2.3 UDP - 500 KB - 32 Mbps - Uplink

```
iperf 3.7
Linux tms 5.13.0-40-generic #45~20.04.1-Ubuntu SMP Mon Apr 4 09:38:31 UTC 2022 x86_64
Time: Sat, 26 Nov 2022 14:56:31 GMT
Connecting to host 192.168.8.20, port 5201
      Cookie: mkxsie5e5ekm4xmdfxstuxhji2cqg3mr3zww
      Target Bitrate: 32000000
[  6] local 192.168.8.3 port 48594 connected to 192.168.8.20 port 5201
Starting Test: protocol: UDP, 1 streams, 500 byte blocks, omitting 0 seconds, 10 second test, tos 0
[ ID] Interval           Transfer     Bitrate         Total Datagrams
[  6]   0.00-1.00   sec  3.81 MBytes  32.0 Mbits/sec  7994
[  6]   1.00-2.00   sec  3.81 MBytes  32.0 Mbits/sec  8000
[  6]   2.00-3.00   sec  3.81 MBytes  32.0 Mbits/sec  8000
[  6]   3.00-4.00   sec  3.81 MBytes  32.0 Mbits/sec  8000
[  6]   4.00-5.00   sec  3.81 MBytes  32.0 Mbits/sec  8000
[  6]   5.00-6.00   sec  3.81 MBytes  32.0 Mbits/sec  8000
[  6]   6.00-7.00   sec  3.81 MBytes  32.0 Mbits/sec  8000
[  6]   7.00-8.00   sec  3.81 MBytes  32.0 Mbits/sec  8000
[  6]   8.00-9.00   sec  3.81 MBytes  32.0 Mbits/sec  8000
[  6]   9.00-10.00  sec  3.81 MBytes  32.0 Mbits/sec  8000
- - - - - - - - - - - - - - - - - - - - - - - - -
Test Complete. Summary Results:
[ ID] Interval           Transfer     Bitrate         Jitter    Lost/Total Datagrams
[  6]   0.00-10.00  sec  38.1 MBytes  32.0 Mbits/sec  0.000 ms  0/79994 (0%)  sender
[  6]   0.00-10.00  sec  38.1 MBytes  32.0 Mbits/sec  0.190 ms  26/79994 (0.033%)  receiver
CPU Utilization: local/sender 6.1% (6.1%u/0.0%s), remote/receiver 6.6% (0.7%u/5.9%s)

iperf Done.
```

A.2.4 UDP - 1448 KB - 90 Mbps - Uplink

```
iperf 3.7
Linux tms 5.13.0-40-generic #45~20.04.1-Ubuntu SMP Mon Apr 4 09:38:31 UTC 2022 x86_64
Time: Sat, 26 Nov 2022 14:55:15 GMT
Connecting to host 192.168.8.20, port 5201
      Cookie: whrytxg7dy5uqtlfyezga4eydelc6jlbfbdu
      Target Bitrate: 90000000
[  6] local 192.168.8.3 port 46344 connected to 192.168.8.20 port 5201
Starting Test: protocol: UDP, 1 streams, 1448 byte blocks, omitting 0 seconds, 10 second test, tos 0
[ ID] Interval           Transfer     Bitrate         Total Datagrams
[  6]   0.00-1.00   sec  10.7 MBytes  89.9 Mbits/sec  7763
[  6]   1.00-2.00   sec  10.7 MBytes  90.0 Mbits/sec  7770
[  6]   2.00-3.00   sec  10.7 MBytes  90.0 Mbits/sec  7769
[  6]   3.00-4.00   sec  10.7 MBytes  90.0 Mbits/sec  7769
[  6]   4.00-5.00   sec  10.7 MBytes  90.0 Mbits/sec  7770
[  6]   5.00-6.00   sec  10.7 MBytes  90.0 Mbits/sec  7769
[  6]   6.00-7.00   sec  10.7 MBytes  90.0 Mbits/sec  7769
[  6]   7.00-8.00   sec  10.7 MBytes  90.0 Mbits/sec  7770
[  6]   8.00-9.00   sec  10.7 MBytes  90.0 Mbits/sec  7769
[  6]   9.00-10.00  sec  10.7 MBytes  90.0 Mbits/sec  7769
- - - - - - - - - - - - - - - - - - - - - - - - -
Test Complete. Summary Results:
[ ID] Interval           Transfer     Bitrate         Jitter    Lost/Total Datagrams
[  6]   0.00-10.00  sec   107 MBytes  90.0 Mbits/sec  0.000 ms  0/77687 (0%)  sender
[  6]   0.00-10.00  sec   107 MBytes  89.9 Mbits/sec  0.200 ms  65/77687 (0.084%)  receiver
CPU Utilization: local/sender 6.4% (6.4%u/0.0%s), remote/receiver 7.3% (0.6%u/6.7%s)

iperf Done.
```

A.2.5 UDP - 64 KB - 4.1 Mbps - Downlink

```
iperf 3.7
Linux tms 5.13.0-40-generic #45~20.04.1-Ubuntu SMP Mon Apr 4 09:38:31 UTC 2022 x86_64
Time: Sat, 26 Nov 2022 15:03:45 GMT
Connecting to host 192.168.8.20, port 5201
Reverse mode, remote host 192.168.8.20 is sending
      Cookie: qcbktbkaxbmc2l3vhfv7uckrnqbyojwf2luo
      Target Bitrate: 4099999
[  6] local 192.168.8.3 port 39203 connected to 192.168.8.20 port 5201
Starting Test: protocol: UDP, 1 streams, 64 byte blocks, omitting 0 seconds, 10 second test, tos 0
[ ID] Interval           Transfer     Bitrate         Jitter    Lost/Total Datagrams
[  6]   0.00-1.00   sec   500 KBytes  4.09 Mbits/sec  0.160 ms  0/7995 (0%)
[  6]   1.00-2.00   sec   501 KBytes  4.10 Mbits/sec  0.133 ms  0/8010 (0%)
[  6]   2.00-3.00   sec   501 KBytes  4.10 Mbits/sec  0.139 ms  0/8010 (0%)
[  6]   3.00-4.00   sec   500 KBytes  4.10 Mbits/sec  0.182 ms  0/7999 (0%)
[  6]   4.00-5.00   sec   500 KBytes  4.10 Mbits/sec  0.154 ms  0/8004 (0%)
[  6]   5.00-6.00   sec   500 KBytes  4.09 Mbits/sec  0.145 ms  0/7998 (0%)
[  6]   6.00-7.00   sec   500 KBytes  4.10 Mbits/sec  0.150 ms  0/8003 (0%)
[  6]   7.00-8.00   sec   501 KBytes  4.10 Mbits/sec  0.164 ms  0/8011 (0%)
[  6]   8.00-9.00   sec   500 KBytes  4.10 Mbits/sec  0.147 ms  0/7999 (0%)
[  6]   9.00-10.00  sec   499 KBytes  4.09 Mbits/sec  0.152 ms  0/7985 (0%)
- - - - - - - - - - - - - - - - - - - - - - - - -
Test Complete. Summary Results:
[ ID] Interval           Transfer     Bitrate         Jitter    Lost/Total Datagrams
[  6]   0.00-10.00  sec  4.89 MBytes  4.10 Mbits/sec  0.000 ms  0/80082 (0%)  sender
[  6]   0.00-10.00  sec  4.88 MBytes  4.10 Mbits/sec  0.152 ms  0/80014 (0%)  receiver

iperf Done.
```

A.2.6 UDP - 100 KB - 6.4 Mbps - Downlink

```
iperf 3.7
Linux tms 5.13.0-40-generic #45~20.04.1-Ubuntu SMP Mon Apr 4 09:38:31 UTC 2022 x86_64
Time: Sat, 26 Nov 2022 15:03:02 GMT
```

```
Connecting to host 192.168.8.20, port 5201
Reverse mode, remote host 192.168.8.20 is sending
      Cookie: uj2leieknlwprebdqi6a6kb5kn5s6gpglkyy
      Target Bitrate: 6400000
[  6] local 192.168.8.3 port 44747 connected to 192.168.8.20 port 5201
Starting Test: protocol: UDP, 1 streams, 100 byte blocks, omitting 0 seconds, 10 second test, tos 0
[ ID] Interval           Transfer     Bitrate         Jitter    Lost/Total Datagrams
[  6]   0.00-1.00   sec   781 KBytes  6.39 Mbits/sec  0.182 ms  0/7994 (0%)
[  6]   1.00-2.00   sec   780 KBytes  6.39 Mbits/sec  0.210 ms  0/7989 (0%)
[  6]   2.00-3.00   sec   781 KBytes  6.40 Mbits/sec  0.158 ms  0/8001 (0%)
[  6]   3.00-4.00   sec   781 KBytes  6.40 Mbits/sec  0.191 ms  0/7998 (0%)
[  6]   4.00-5.00   sec   782 KBytes  6.41 Mbits/sec  0.145 ms  0/8011 (0%)
[  6]   5.00-6.00   sec   782 KBytes  6.40 Mbits/sec  0.181 ms  0/8003 (0%)
[  6]   6.00-7.00   sec   779 KBytes  6.38 Mbits/sec  0.147 ms  0/7977 (0%)
[  6]   7.00-8.00   sec   783 KBytes  6.41 Mbits/sec  0.188 ms  0/8016 (0%)
[  6]   8.00-9.00   sec   781 KBytes  6.40 Mbits/sec  0.159 ms  0/7997 (0%)
[  6]   9.00-10.00  sec   779 KBytes  6.38 Mbits/sec  0.176 ms  0/7979 (0%)
- - - - - - - - - - - - - - - - - - - - - - - - -
Test Complete. Summary Results:
[ ID] Interval           Transfer     Bitrate         Jitter    Lost/Total Datagrams
[  6]   0.00-10.00  sec  7.63 MBytes  6.40 Mbits/sec  0.000 ms  0/80000 (0%)  sender
[  6]   0.00-10.00  sec  7.63 MBytes  6.40 Mbits/sec  0.176 ms  0/79965 (0%)  receiver

iperf Done.
```

A.2.7 UDP - 500 KB - 32 Mbps - Downlink

```
iperf 3.7
Linux tms 5.13.0-40-generic #45~20.04.1-Ubuntu SMP Mon Apr 4 09:38:31 UTC 2022 x86_64
Time: Sat, 26 Nov 2022 15:02:21 GMT
Connecting to host 192.168.8.20, port 5201
Reverse mode, remote host 192.168.8.20 is sending
      Cookie: gdustswbxenkoaw2lyrauo54dcrjtskqyusy
      Target Bitrate: 32000000
[  6] local 192.168.8.3 port 45426 connected to 192.168.8.20 port 5201
Starting Test: protocol: UDP, 1 streams, 500 byte blocks, omitting 0 seconds, 10 second test, tos 0
[ ID] Interval           Transfer     Bitrate         Jitter    Lost/Total Datagrams
[  6]   0.00-1.00   sec  3.81 MBytes  31.9 Mbits/sec  0.166 ms  0/7981 (0%)
[  6]   1.00-2.00   sec  3.80 MBytes  31.9 Mbits/sec  0.184 ms  0/7972 (0%)
[  6]   2.00-3.00   sec  3.80 MBytes  31.9 Mbits/sec  0.183 ms  0/7979 (0%)
[  6]   3.00-4.00   sec  3.80 MBytes  31.9 Mbits/sec  0.122 ms  0/7976 (0%)
[  6]   4.00-5.00   sec  3.80 MBytes  31.9 Mbits/sec  0.125 ms  0/7975 (0%)
[  6]   5.00-6.00   sec  3.80 MBytes  31.9 Mbits/sec  0.162 ms  0/7978 (0%)
[  6]   6.00-7.00   sec  3.80 MBytes  31.9 Mbits/sec  0.087 ms  0/7969 (0%)
[  6]   7.00-8.00   sec  3.80 MBytes  31.9 Mbits/sec  0.095 ms  0/7974 (0%)
[  6]   8.00-9.00   sec  3.80 MBytes  31.9 Mbits/sec  0.092 ms  0/7978 (0%)
[  6]   9.00-10.00  sec  3.81 MBytes  31.9 Mbits/sec  0.154 ms  0/7980 (0%)
- - - - - - - - - - - - - - - - - - - - - - - - -
Test Complete. Summary Results:
[ ID] Interval           Transfer     Bitrate         Jitter    Lost/Total Datagrams
[  6]   0.00-10.00  sec  38.1 MBytes  31.9 Mbits/sec  0.000 ms  0/79839 (0%)  sender
[  6]   0.00-10.00  sec  38.0 MBytes  31.9 Mbits/sec  0.154 ms  0/79762 (0%)  receiver

iperf Done.
```

A.2.8 UDP - 1448 KB - 90 Mbps - Downlink

```
iperf 3.7
Linux tms 5.13.0-40-generic #45~20.04.1-Ubuntu SMP Mon Apr 4 09:38:31 UTC 2022 x86_64
Time: Sat, 26 Nov 2022 15:01:08 GMT
Connecting to host 192.168.8.20, port 5201
Reverse mode, remote host 192.168.8.20 is sending
      Cookie: emiwmmgwo2skod4fw6pietnrpexwc6t45nzq
      Target Bitrate: 90000000
[  6] local 192.168.8.3 port 35457 connected to 192.168.8.20 port 5201
Starting Test: protocol: UDP, 1 streams, 1448 byte blocks, omitting 0 seconds, 10 second test, tos 0
[ ID] Interval           Transfer     Bitrate         Jitter    Lost/Total Datagrams
[  6]   0.00-1.00   sec  10.7 MBytes  89.9 Mbits/sec  0.140 ms  0/7765 (0%)
[  6]   1.00-2.00   sec  10.7 MBytes  90.0 Mbits/sec  0.156 ms  0/7767 (0%)
[  6]   2.00-3.00   sec  10.7 MBytes  90.0 Mbits/sec  0.141 ms  0/7770 (0%)
[  6]   3.00-4.00   sec  10.7 MBytes  90.0 Mbits/sec  0.169 ms  0/7769 (0%)
[  6]   4.00-5.00   sec  10.7 MBytes  90.0 Mbits/sec  0.160 ms  0/7771 (0%)
[  6]   5.00-6.00   sec  10.7 MBytes  90.0 Mbits/sec  0.146 ms  0/7769 (0%)
[  6]   6.00-7.00   sec  10.7 MBytes  90.0 Mbits/sec  0.151 ms  0/7769 (0%)
[  6]   7.00-8.00   sec  10.7 MBytes  90.0 Mbits/sec  0.140 ms  0/7769 (0%)
[  6]   8.00-9.00   sec  10.7 MBytes  90.0 Mbits/sec  0.152 ms  0/7770 (0%)
[  6]   9.00-10.00  sec  10.7 MBytes  90.0 Mbits/sec  0.154 ms  0/7769 (0%)
- - - - - - - - - - - - - - - - - - - - - - - - -
Test Complete. Summary Results:
[ ID] Interval           Transfer     Bitrate         Jitter    Lost/Total Datagrams
[  6]   0.00-10.00  sec   107 MBytes  90.0 Mbits/sec  0.000 ms  0/77692 (0%)  sender
[  6]   0.00-10.00  sec   107 MBytes  90.0 Mbits/sec  0.154 ms  0/77688 (0%)  receiver

iperf Done.
```

A.2.9 UDP - 64 KB - 4.1 Mbps - Bidirectional

```
iperf 3.7
Linux tms 5.13.0-40-generic #45~20.04.1-Ubuntu SMP Mon Apr 4 09:38:31 UTC 2022 x86_64
Time: Sat, 26 Nov 2022 15:10:24 GMT
Connecting to host 192.168.8.20, port 5201
      Cookie: unpgqopesssxryt3wknjy334kcpkwupmehgj
      Target Bitrate: 4099999
[  6] local 192.168.8.3 port 33994 connected to 192.168.8.20 port 5201
[  8] local 192.168.8.3 port 42588 connected to 192.168.8.20 port 5201
Starting Test: protocol: UDP, 1 streams, 64 byte blocks, omitting 0 seconds, 10 second test, tos 0
[ ID][Role] Interval           Transfer     Bitrate         Jitter    Lost/Total Datagrams
[  6][TX-C]   0.00-1.00   sec   500 KBytes  4.10 Mbits/sec            8001
[  8][RX-C]   0.00-1.00   sec   501 KBytes  4.10 Mbits/sec  0.089 ms  0/8010 (0%)
[  6][TX-C]   1.00-2.00   sec   500 KBytes  4.10 Mbits/sec            8007
[  8][RX-C]   1.00-2.00   sec   500 KBytes  4.10 Mbits/sec  0.097 ms  0/8006 (0%)
[  6][TX-C]   2.00-3.00   sec   500 KBytes  4.10 Mbits/sec            8008
[  8][RX-C]   2.00-3.00   sec   500 KBytes  4.10 Mbits/sec  0.118 ms  0/8006 (0%)
[  6][TX-C]   3.00-4.00   sec   500 KBytes  4.10 Mbits/sec            8008
[  8][RX-C]   3.00-4.00   sec   500 KBytes  4.10 Mbits/sec  0.114 ms  0/8008 (0%)
[  6][TX-C]   4.00-5.00   sec   500 KBytes  4.10 Mbits/sec            8008
[  8][RX-C]   4.00-5.00   sec   500 KBytes  4.10 Mbits/sec  0.125 ms  0/8008 (0%)
[  6][TX-C]   5.00-6.00   sec   500 KBytes  4.10 Mbits/sec            8008
[  8][RX-C]   5.00-6.00   sec   501 KBytes  4.10 Mbits/sec  0.124 ms  0/8013 (0%)
[  6][TX-C]   6.00-7.00   sec   500 KBytes  4.10 Mbits/sec            8007
[  8][RX-C]   6.00-7.00   sec   500 KBytes  4.10 Mbits/sec  0.122 ms  0/8008 (0%)
[  6][TX-C]   7.00-8.00   sec   500 KBytes  4.10 Mbits/sec            8008
[  8][RX-C]   7.00-8.00   sec   500 KBytes  4.10 Mbits/sec  0.114 ms  0/8001 (0%)
[  6][TX-C]   8.00-9.00   sec   500 KBytes  4.10 Mbits/sec            8008
[  8][RX-C]   8.00-9.00   sec   501 KBytes  4.10 Mbits/sec  0.105 ms  0/8016 (0%)
[  6][TX-C]   9.00-10.00  sec   500 KBytes  4.10 Mbits/sec            8008
[  8][RX-C]   9.00-10.00  sec   500 KBytes  4.10 Mbits/sec  0.106 ms  0/8004 (0%)
- - - - - - - - - - - - - - - - - - - - - - - - -
Test Complete. Summary Results:
[ ID][Role] Interval           Transfer     Bitrate         Jitter    Lost/Total Datagrams
[  6][TX-C]   0.00-10.00  sec  4.89 MBytes  4.10 Mbits/sec  0.000 ms  0/80071 (0%)  sender
[  6][TX-C]   0.00-10.00  sec  4.89 MBytes  4.10 Mbits/sec  0.133 ms  19/80071 (0.024%)
receiver
CPU Utilization: local/sender 100.0% (22.1%u/77.8%s), remote/receiver 25.1% (1.4%u/23.7%s)
CPU Utilization: local/receiver 100.0% (22.1%u/77.8%s), remote/sender 25.1% (1.4%u/23.7%s)
[  8][RX-C]   0.00-10.00  sec  4.89 MBytes  4.10 Mbits/sec  0.000 ms  0/80090 (0%)  sender
[  8][RX-C]   0.00-10.00  sec  4.89 MBytes  4.10 Mbits/sec  0.106 ms  0/80080 (0%)  receiver

iperf Done.
```

A.2.10 UDP - 100 KB - 6.4 Mbps - Bidirectional

```
iperf 3.7
Linux tms 5.13.0-40-generic #45~20.04.1-Ubuntu SMP Mon Apr 4 09:38:31 UTC 2022 x86_64
Time: Sat, 26 Nov 2022 15:09:44 GMT
Connecting to host 192.168.8.20, port 5201
      Cookie: 3u2rvt4elbes7szl5igbtiiwhyd7rdbikb7z
      Target Bitrate: 6400000
[  6] local 192.168.8.3 port 53484 connected to 192.168.8.20 port 5201
[  8] local 192.168.8.3 port 44144 connected to 192.168.8.20 port 5201
Starting Test: protocol: UDP, 1 streams, 100 byte blocks, omitting 0 seconds, 10 second test, tos 0
[ ID][Role] Interval           Transfer     Bitrate         Jitter    Lost/Total Datagrams
[  6][TX-C]   0.00-1.00   sec   781 KBytes  6.39 Mbits/sec            7993
[  8][RX-C]   0.00-1.00   sec   781 KBytes  6.40 Mbits/sec  0.095 ms  0/7999 (0%)
[  6][TX-C]   1.00-2.00   sec   781 KBytes  6.40 Mbits/sec            8000
[  8][RX-C]   1.00-2.00   sec   781 KBytes  6.40 Mbits/sec  0.103 ms  0/7996 (0%)
[  6][TX-C]   2.00-3.00   sec   781 KBytes  6.40 Mbits/sec            8000
[  8][RX-C]   2.00-3.00   sec   782 KBytes  6.40 Mbits/sec  0.109 ms  0/8003 (0%)
[  6][TX-C]   3.00-4.00   sec   781 KBytes  6.40 Mbits/sec            8000
[  8][RX-C]   3.00-4.00   sec   781 KBytes  6.40 Mbits/sec  0.107 ms  0/7995 (0%)
[  6][TX-C]   4.00-5.00   sec   781 KBytes  6.40 Mbits/sec            8000
[  8][RX-C]   4.00-5.00   sec   781 KBytes  6.40 Mbits/sec  0.093 ms  0/8001 (0%)
[  6][TX-C]   5.00-6.00   sec   781 KBytes  6.40 Mbits/sec            8000
[  8][RX-C]   5.00-6.00   sec   782 KBytes  6.40 Mbits/sec  0.103 ms  0/8005 (0%)
[  6][TX-C]   6.00-7.00   sec   781 KBytes  6.40 Mbits/sec            8000
[  8][RX-C]   6.00-7.00   sec   781 KBytes  6.40 Mbits/sec  0.102 ms  0/7994 (0%)
[  6][TX-C]   7.00-8.00   sec   781 KBytes  6.40 Mbits/sec            8000
[  8][RX-C]   7.00-8.00   sec   782 KBytes  6.40 Mbits/sec  0.103 ms  0/8006 (0%)
[  6][TX-C]   8.00-9.00   sec   781 KBytes  6.40 Mbits/sec            8000
[  8][RX-C]   8.00-9.00   sec   781 KBytes  6.40 Mbits/sec  0.118 ms  0/7994 (0%)
[  6][TX-C]   9.00-10.00  sec   781 KBytes  6.40 Mbits/sec            8000
[  8][RX-C]   9.00-10.00  sec   782 KBytes  6.40 Mbits/sec  0.094 ms  0/8005 (0%)
- - - - - - - - - - - - - - - - - - - - - - - - -
Test Complete. Summary Results:
[ ID][Role] Interval           Transfer     Bitrate         Jitter    Lost/Total Datagrams
[  6][TX-C]   0.00-10.00  sec  7.63 MBytes  6.40 Mbits/sec  0.000 ms  0/79993 (0%)  sender
[  6][TX-C]   0.00-10.00  sec  7.63 MBytes  6.40 Mbits/sec  0.102 ms  16/79993 (0.02%)
receiver
CPU Utilization: local/sender 100.0% (24.7%u/75.3%s), remote/receiver 21.3% (2.0%u/19.3%s)
CPU Utilization: local/receiver 100.0% (24.7%u/75.3%s), remote/sender 21.3% (2.0%u/19.3%s)
[  8][RX-C]   0.00-10.00  sec  7.63 MBytes  6.40 Mbits/sec  0.000 ms  0/80006 (0%)  sender
[  8][RX-C]   0.00-10.00  sec  7.63 MBytes  6.40 Mbits/sec  0.094 ms  0/79998 (0%)  receiver

iperf Done.
```

A.2.11 UDP - 500 KB - 32 Mbps - Bidirectional

```
iperf 3.7
Linux tms 5.13.0-40-generic #45~20.04.1-Ubuntu SMP Mon Apr 4 09:38:31 UTC 2022 x86_64
Time: Sat, 26 Nov 2022 15:08:58 GMT
Connecting to host 192.168.8.20, port 5201
      Cookie: shcj2vr7v4mawc4xqewnjtsqehgddbcjnn4t
      Target Bitrate: 32000000
[  6] local 192.168.8.3 port 57779 connected to 192.168.8.20 port 5201
[  8] local 192.168.8.3 port 34737 connected to 192.168.8.20 port 5201
Starting Test: protocol: UDP, 1 streams, 500 byte blocks, omitting 0 seconds, 10 second test, tos 0
[ ID][Role] Interval           Transfer     Bitrate         Jitter    Lost/Total Datagrams
[  6][TX-C]   0.00-1.00   sec  3.81 MBytes  32.0 Mbits/sec            7993
[  8][RX-C]   0.00-1.00   sec  3.81 MBytes  32.0 Mbits/sec  0.122 ms  0/7996 (0%)
[  6][TX-C]   1.00-2.00   sec  3.81 MBytes  32.0 Mbits/sec            8000
[  8][RX-C]   1.00-2.00   sec  3.81 MBytes  32.0 Mbits/sec  0.102 ms  0/7999 (0%)
[  6][TX-C]   2.00-3.00   sec  3.81 MBytes  32.0 Mbits/sec            8000
[  8][RX-C]   2.00-3.00   sec  3.81 MBytes  32.0 Mbits/sec  0.110 ms  0/7998 (0%)
[  6][TX-C]   3.00-4.00   sec  3.81 MBytes  32.0 Mbits/sec            8000
[  8][RX-C]   3.00-4.00   sec  3.82 MBytes  32.0 Mbits/sec  0.114 ms  0/8002 (0%)
[  6][TX-C]   4.00-5.00   sec  3.81 MBytes  32.0 Mbits/sec            8000
[  8][RX-C]   4.00-5.00   sec  3.82 MBytes  32.0 Mbits/sec  0.109 ms  0/8005 (0%)
[  6][TX-C]   5.00-6.00   sec  3.81 MBytes  32.0 Mbits/sec            8000
[  8][RX-C]   5.00-6.00   sec  3.81 MBytes  32.0 Mbits/sec  0.123 ms  0/7993 (0%)
[  6][TX-C]   6.00-7.00   sec  3.81 MBytes  32.0 Mbits/sec            8000
[  8][RX-C]   6.00-7.00   sec  3.82 MBytes  32.0 Mbits/sec  0.118 ms  0/8004 (0%)
[  6][TX-C]   7.00-8.00   sec  3.81 MBytes  32.0 Mbits/sec            8000
[  8][RX-C]   7.00-8.00   sec  3.82 MBytes  32.0 Mbits/sec  0.113 ms  0/8001 (0%)
[  6][TX-C]   8.00-9.00   sec  3.81 MBytes  32.0 Mbits/sec            8000
[  8][RX-C]   8.00-9.00   sec  3.81 MBytes  32.0 Mbits/sec  0.118 ms  0/7995 (0%)
[  6][TX-C]   9.00-10.00  sec  3.81 MBytes  32.0 Mbits/sec            8000
[  8][RX-C]   9.00-10.00  sec  3.82 MBytes  32.0 Mbits/sec  0.114 ms  0/8004 (0%)
- - - - - - - - - - - - - - - - - - - - - - - - -
Test Complete. Summary Results:
[ ID][Role] Interval           Transfer     Bitrate         Jitter    Lost/Total Datagrams
[  6][TX-C]   0.00-10.00  sec  38.1 MBytes  32.0 Mbits/sec  0.000 ms  0/79993 (0%)  sender
[  6][TX-C]   0.00-10.00  sec  38.0 MBytes  31.9 Mbits/sec  0.127 ms  263/79993 (0.33%)
receiver
CPU Utilization: local/sender 100.0% (24.1%u/75.8%s), remote/receiver 12.3% (1.1%u/11.3%s)
CPU Utilization: local/receiver 100.0% (24.1%u/75.8%s), remote/sender 12.3% (1.1%u/11.3%s)
[  8][RX-C]   0.00-10.00  sec  38.1 MBytes  32.0 Mbits/sec  0.000 ms  0/80006 (0%)  sender
[  8][RX-C]   0.00-10.00  sec  38.1 MBytes  32.0 Mbits/sec  0.114 ms  0/79997 (0%)  receiver

iperf Done.
```

A.2.12 UDP - 1448 KB - 90 Mbps - Bidirectional

```
iperf 3.7
Linux tms 5.13.0-40-generic #45~20.04.1-Ubuntu SMP Mon Apr 4 09:38:31 UTC 2022 x86_64
Time: Sat, 26 Nov 2022 15:07:37 GMT
Connecting to host 192.168.8.20, port 5201
      Cookie: x3bkzhbsbdrn34yvxbgoo6veqgqv5nyewlhg
      Target Bitrate: 90000000
[  6] local 192.168.8.3 port 39110 connected to 192.168.8.20 port 5201
[  8] local 192.168.8.3 port 60876 connected to 192.168.8.20 port 5201
Starting Test: protocol: UDP, 1 streams, 1448 byte blocks, omitting 0 seconds, 10 second test, tos 0
[ ID][Role] Interval           Transfer     Bitrate         Jitter    Lost/Total Datagrams
[  6][TX-C]   0.00-1.00   sec  10.7 MBytes  89.9 Mbits/sec            7762
[  8][RX-C]   0.00-1.00   sec  10.3 MBytes  86.6 Mbits/sec  0.170 ms  0/7473 (0%)
[  6][TX-C]   1.00-2.00   sec  10.7 MBytes  90.0 Mbits/sec            7770
[  8][RX-C]   1.00-2.00   sec  10.3 MBytes  86.7 Mbits/sec  0.135 ms  0/7487 (0%)
[  6][TX-C]   2.00-3.00   sec  10.7 MBytes  90.0 Mbits/sec            7769
[  8][RX-C]   2.00-3.00   sec  10.3 MBytes  86.4 Mbits/sec  0.079 ms  0/7461 (0%)
[  6][TX-C]   3.00-4.00   sec  10.7 MBytes  90.0 Mbits/sec            7769
[  8][RX-C]   3.00-4.00   sec  10.3 MBytes  86.5 Mbits/sec  0.167 ms  0/7465 (0%)
[  6][TX-C]   4.00-5.00   sec  10.7 MBytes  90.0 Mbits/sec            7770
[  8][RX-C]   4.00-5.00   sec  10.3 MBytes  86.8 Mbits/sec  0.170 ms  0/7494 (0%)
[  6][TX-C]   5.00-6.00   sec  10.7 MBytes  90.0 Mbits/sec            7769
[  8][RX-C]   5.00-6.00   sec  10.3 MBytes  86.8 Mbits/sec  0.108 ms  0/7493 (0%)
[  6][TX-C]   6.00-7.00   sec  10.7 MBytes  90.0 Mbits/sec            7769
[  8][RX-C]   6.00-7.00   sec  10.3 MBytes  86.5 Mbits/sec  0.078 ms  0/7468 (0%)
[  6][TX-C]   7.00-8.00   sec  10.7 MBytes  90.0 Mbits/sec            7770
[  8][RX-C]   7.00-8.00   sec  10.4 MBytes  86.9 Mbits/sec  0.085 ms  0/7504 (0%)
[  6][TX-C]   8.00-9.00   sec  10.7 MBytes  90.0 Mbits/sec            7769
[  8][RX-C]   8.00-9.00   sec  10.3 MBytes  86.8 Mbits/sec  0.090 ms  0/7493 (0%)
[  6][TX-C]   9.00-10.00  sec  10.7 MBytes  90.0 Mbits/sec            7769
[  8][RX-C]   9.00-10.00  sec  10.3 MBytes  86.6 Mbits/sec  0.171 ms  0/7477 (0%)
- - - - - - - - - - - - - - - - - - - - - - - - -
Test Complete. Summary Results:
[ ID][Role] Interval           Transfer     Bitrate         Jitter    Lost/Total Datagrams
[  6][TX-C]   0.00-10.00  sec   107 MBytes  90.0 Mbits/sec  0.000 ms  0/77686 (0%)  sender
[  6][TX-C]   0.00-10.08  sec   106 MBytes  88.1 Mbits/sec  0.202 ms  1064/77686 (1.4%)
receiver
CPU Utilization: local/sender 100.0% (23.2%u/76.8%s), remote/receiver 7.6% (0.5%u/7.1%s)
CPU Utilization: local/receiver 100.0% (23.2%u/76.8%s), remote/sender 7.6% (0.5%u/7.1%s)
```

```
[  8][RX-C]   0.00-10.00  sec   104 MBytes  87.4 Mbits/sec  0.000 ms  0/75451 (0%)  sender
[  8][RX-C]   0.00-10.08  sec   103 MBytes  86.0 Mbits/sec  0.171 ms  0/74815 (0%)  receiver

iperf Done.
```

B Ping Results

B.1 Latency Uplink 10 s

```
PING 192.168.8.3 (192.168.8.3) 56(84) bytes of data.
64 bytes from 192.168.8.3: icmp_seq=1 ttl=64 time=1.03 ms
64 bytes from 192.168.8.3: icmp_seq=2 ttl=64 time=0.945 ms
64 bytes from 192.168.8.3: icmp_seq=3 ttl=64 time=0.858 ms
64 bytes from 192.168.8.3: icmp_seq=4 ttl=64 time=0.996 ms
64 bytes from 192.168.8.3: icmp_seq=5 ttl=64 time=0.721 ms
64 bytes from 192.168.8.3: icmp_seq=6 ttl=64 time=0.892 ms
64 bytes from 192.168.8.3: icmp_seq=7 ttl=64 time=0.893 ms
64 bytes from 192.168.8.3: icmp_seq=8 ttl=64 time=0.718 ms
64 bytes from 192.168.8.3: icmp_seq=9 ttl=64 time=0.918 ms
64 bytes from 192.168.8.3: icmp_seq=10 ttl=64 time=0.956 ms

--- 192.168.8.3 ping statistics ---
10 packets transmitted, 10 received, 0% packet loss, time 9126ms
rtt min/avg/max/mdev = 0.718/0.892/1.026/0.098 ms
```

B.2 Latency Downlink 10 s

```
PING 192.168.8.20 (192.168.8.20) 56(84) bytes of data.
64 bytes from 192.168.8.20: icmp_seq=1 ttl=64 time=0.747 ms
64 bytes from 192.168.8.20: icmp_seq=2 ttl=64 time=0.960 ms
64 bytes from 192.168.8.20: icmp_seq=3 ttl=64 time=1.06 ms
64 bytes from 192.168.8.20: icmp_seq=4 ttl=64 time=0.904 ms
64 bytes from 192.168.8.20: icmp_seq=5 ttl=64 time=1.12 ms
64 bytes from 192.168.8.20: icmp_seq=6 ttl=64 time=0.900 ms
64 bytes from 192.168.8.20: icmp_seq=7 ttl=64 time=0.906 ms
64 bytes from 192.168.8.20: icmp_seq=8 ttl=64 time=1.13 ms
64 bytes from 192.168.8.20: icmp_seq=9 ttl=64 time=0.937 ms
64 bytes from 192.168.8.20: icmp_seq=10 ttl=64 time=0.785 ms

--- 192.168.8.20 ping statistics ---
10 packets transmitted, 10 received, 0% packet loss, time 9040ms
rtt min/avg/max/mdev = 0.747/0.944/1.129/0.121 m
```

B.3 Flooding - 64 B - Downlink

```
PING 192.168.8.20 (192.168.8.20) 56(84) bytes of data.
.
--- 192.168.8.20 ping statistics ---
79582 packets transmitted, 79581 received, 0.00125657% packet loss, time 60000ms
rtt min/avg/max/mdev = 0.382/0.707/15.254/0.224 ms, pipe 2, ipg/ewma 0.753/0.683 ms
```

B.4 Flooding - 100 B - Downlink

```
PING 192.168.8.20 (192.168.8.20) 92(120) bytes of data.
..
--- 192.168.8.20 ping statistics ---
79292 packets transmitted, 79290 received, 0.00252232% packet loss, time 59999ms
rtt min/avg/max/mdev = 0.427/0.710/15.604/0.223 ms, pipe 2, ipg/ewma 0.756/0.724 ms
```

B.5 Flooding - 500 B - Downlink

```
PING 192.168.8.20 (192.168.8.20) 492(520) bytes of data.
.
--- 192.168.8.20 ping statistics ---
77670 packets transmitted, 77669 received, 0.0012875% packet loss, time 60000ms
rtt min/avg/max/mdev = 0.475/0.720/15.585/0.238 ms, pipe 2, ipg/ewma 0.772/0.718 ms
```

B.6 Flooding - 1448 B - Downlink

```
PING 192.168.8.20 (192.168.8.20) 1440(1468) bytes of data.

--- 192.168.8.20 ping statistics ---
47034 packets transmitted, 47034 received, 0% packet loss, time 59999ms
rtt min/avg/max/mdev = 0.908/1.159/15.347/0.292 ms, pipe 2, ipg/ewma 1.275/1.163 ms
```

B.7 Flooding - 64 B - Uplink

```
PING 192.168.8.3 (192.168.8.3) 56(84) bytes of data.
.
--- 192.168.8.3 ping statistics ---
66223 packets transmitted, 66222 received, 0.00151005% packet loss, time 59999ms
rtt min/avg/max/mdev = 0.453/0.748/15.278/0.258 ms, pipe 2, ipg/ewma 0.906/0.779 ms
```

B.8 Flooding - 100 B - Uplink

```
PING 192.168.8.3 (192.168.8.3) 92(120) bytes of data.
.
--- 192.168.8.3 ping statistics ---
63990 packets transmitted, 63989 received, 0.00156274% packet loss, time 59999ms
rtt min/avg/max/mdev = 0.449/0.774/28.881/0.285 ms, pipe 2, ipg/ewma 0.937/0.820 ms
```

B.9 Flooding - 500 B - Uplink

```
PING 192.168.8.3 (192.168.8.3) 492(520) bytes of data.
.
--- 192.168.8.3 ping statistics ---
56451 packets transmitted, 56450 received, 0.00177145% packet loss, time 59999ms
rtt min/avg/max/mdev = 0.551/0.888/15.126/0.255 ms, pipe 2, ipg/ewma 1.062/0.838 ms
```

B.10 Flooding - 1448 B - Uplink

```
PING 192.168.8.3 (192.168.8.3) 1440(1468) bytes of data.

--- 192.168.8.3 ping statistics ---
44269 packets transmitted, 44269 received, 0% packet loss, time 59999ms
rtt min/avg/max/mdev = 0.818/1.158/15.484/0.305 ms, pipe 2, ipg/ewma 1.355/1.114 ms
```