

# **LIMITING PACKET LOSS ON A STRICTLY UNIDIRECTIONAL PHYSICAL DATA DIODE**

by

**G.J. den Besten**

in partial fulfillment of the requirements for the degree of

**Master of Science**  
in Software Engineering

at the Open University of the Netherlands  
Faculty of Science  
Master's Programme in Software Engineering

to be defended publicly on October 27th, 2025 at 14:00 (CET).

Student number: 835333020

Course code: IM9906

Thesis committee:	prof. dr. ir. H.P.E. Vranken	Open Universiteit
	dr. F.M.J. van den Broek	Open Universiteit
	dr. P.G. Story (external member)	Clark University

# ACKNOWLEDGEMENTS

This thesis marks the end of a long journey that began in 1992. After obtaining my Bachelor's degree in Mechanical Engineering and entering the workforce in 1991, I decided to continue my education by taking Computer Science related courses at the Dutch Open Universiteit. Over the past 33 years, I have experienced several long interruptions, but with this work, I am finally completing my studies.

First of all, I would like to thank my supervisor Dr. Fabian van den Broek for his helpful advice and the many Teams sessions that guided me through this project.

I would also like to thank my supervisors within the Dutch Ministry of Defence for providing the necessary conditions to complete this study. I am also grateful to my innovation manager for his valuable suggestions and guidance. Also, a big thank you to Iwan Flaming, who first introduced me to the concept of data diodes back in 2021.

Another person I would like to thank is René Vrolijk for enthusiastically sharing data diode-related information and workshops on the GitHub platform. René, thank you for your input and suggestions.

I also wish to thank Dr. Peter Story (Clark University) for sharing his insights, materials, suggestions, and his valuable feedback throughout this graduation project.

Finally, I would like to express my heartfelt appreciation to my partner Mina for her understanding and the support she has given me throughout this project.

*In loving memory of Marja (1973 - 2019) †*

# SUMMARY

Physical data diodes are a cybersecurity measure that enable unidirectional data transfer between physically separated networks. This one-way communication prevents information from leaking from the destination network back to the source network. Although various commercial solutions are available, relatively few academic publications address data diodes.

One of the main challenges of unidirectional data transfer is the potential loss of transmitted data, also known as packet loss. This study aims to contribute to the existing body of research by laying the groundwork for further investigation into where and when packet loss occurs and how it can be mitigated. This thesis describes an enhancement to the Linux UDPc kernel module, `dd_udp.c`, developed to handle UDP traffic within data diodes more efficiently. In addition, measurement points within the Linux networking stack are selected and other measurement points are implemented in this alternative module, providing greater insight into UDP data transmission.

This study makes use of a simple data diode test setup that is described both on an informative GitHub website and in an academic publication from 2023. This setup makes use of a single workstation with two network interfaces. Tests are conducted by sending files using Netcat and pydiode. The latter tool was introduced in the aforementioned 2023 publication. While Netcat allows for simple data transmission, pydiode enables the use of redundant data transmission. Additional tests are performed with increased UDP receiver socket buffer sizes, modified values for the MTU, and for modified data segment sizes that Netcat and pydiode attempt to send.

The results show that there are no consistent performance or efficiency differences between the two kernel modules. When testing the other factors, it is observed that increasing the UDP receiver socket buffer, when using pydiode, quickly leads to an efficiency of 100 %, whereas this was not always the case in the 2023 publication. Variations in the other tested factors have little impact within the context of this test setup.

Recommendations for future research include modifying the `dd_udp.c` module for use on two separate workstations, thereby turning the module itself into a security measure; conducting test scenarios that more closely resemble real-life situations; experimenting with system settings on which the sender and receiver processes run; and, finally, developing a predictive model to minimize packet loss.

# CONTENTS

<b>List of Figures</b>	<b>vi</b>
<b>List of Tables</b>	<b>vii</b>
<b>1 Introduction</b>	<b>1</b>
1.1 Motivation . . . . .	1
1.2 Research questions . . . . .	2
1.3 Overview . . . . .	3
<b>2 Background</b>	<b>4</b>
2.1 Overview of UDP and the Internet Model . . . . .	4
2.1.1 UDP dataflow through the Internet Model . . . . .	5
2.2 The UDP protocol . . . . .	6
2.2.1 UDP header . . . . .	6
2.2.2 UDP Checksum . . . . .	7
2.3 Data transfer with Jumboframes . . . . .	7
2.3.1 MTU . . . . .	7
2.3.2 Jumboframes . . . . .	8
2.3.3 MTU and the handling of Jumboframes . . . . .	8
2.4 Basic architecture of data diodes . . . . .	9
2.5 Packet Loss . . . . .	9
2.6 Known methods to reduce packet loss . . . . .	10
2.7 Loss ratios . . . . .	11
<b>3 Related Work</b>	<b>12</b>
3.1 Unidirectional datatransfer and packet loss . . . . .	12
3.2 A practical approach . . . . .	13
3.2.1 The Vrolijk setup . . . . .	13
3.2.2 Pydiode: chunks, rate limiting and redundant transmission . . . . .	14
3.3 MTU and efficiency . . . . .	15
<b>4 Research</b>	<b>16</b>
4.1 Method . . . . .	16
4.2 Tests and experiments . . . . .	16
4.2.1 The data diode test setup . . . . .	17
4.2.2 The data diode network testing setup . . . . .	17
4.2.3 Performing tests . . . . .	18
4.2.4 Scripting . . . . .	20
4.2.5 Test output and analysis . . . . .	20
4.2.6 Recapitulation of the test setup . . . . .	21

<b>5</b>	<b>UDP modules</b>	<b>22</b>
5.1	The UDPC kernel module . . . . .	22
5.1.1	Modifying the UDPC module . . . . .	23
5.2	The dd_udp.c module . . . . .	23
5.2.1	Features and capabilities of dd_udp.c . . . . .	23
5.2.2	Functionality omitted in dd_udp.c . . . . .	25
5.2.3	Expectations for the dd_udp.c module . . . . .	25
5.3	Baseline test results for both kernel modules . . . . .	26
<b>6</b>	<b>Measuring points</b>	<b>28</b>
6.1	Relevant measuring points . . . . .	28
6.1.1	File Integrity - Application Layer (Layer 4) . . . . .	28
6.1.2	Sockets - Transport Layer (Layer 3) . . . . .	28
6.1.3	Kernel module - Transport/Internet Layer (Layers 3/2) . . . . .	29
6.1.4	Link Layer (Layer 1) . . . . .	29
6.2	Sending Proxy . . . . .	29
6.2.1	Transport Layer . . . . .	29
6.2.2	dd_udp.c module: Transport (UDP) and Internet (IP) Layers . . . . .	30
6.2.3	Link Layer . . . . .	31
6.3	Receiving Proxy . . . . .	31
6.3.1	Link Layer . . . . .	31
6.3.2	dd_udp.c module: Transport (UDP) and Internet (IP) Layers . . . . .	32
6.3.3	Transport Layer (UDP) . . . . .	32
6.4	Measuring Points and Test Scripting . . . . .	32
6.5	Observations . . . . .	33
6.5.1	The usefulness of UDP checksumming . . . . .	33
6.5.2	InErrors versus RcvbufErrors . . . . .	33
6.5.3	Impact of tcpdump . . . . .	34
6.5.4	SndbufErrors > 0 with pydiode despite correct file reception . . . . .	34
<b>7</b>	<b>Modifying internal parameters of the data diode proxies</b>	<b>35</b>
7.1	The UDP receiving proxy socket buffer . . . . .	35
7.1.1	Observations . . . . .	37
7.2	The MTU values of the data diode proxies . . . . .	37
7.2.1	Observations . . . . .	40
7.3	Adjusting data segments to the maximum MTU value . . . . .	41
7.3.1	Adjusting the Data Segment Size . . . . .	41
7.3.2	Observations . . . . .	42
<b>8</b>	<b>Discussion</b>	<b>44</b>
8.1	Comparing Baseline Results to Previous Research . . . . .	44
8.2	Using a NUC as a Data Diode Testing Platform . . . . .	46
8.2.1	Kubuntu OS . . . . .	46
8.2.2	System Load Limitations . . . . .	46
8.3	dd_udp.c as a Loadable Kernel Module . . . . .	46

<b>9</b>	<b>Conclusion and Future Work</b>	<b>48</b>
9.1	Context . . . . .	48
9.2	Conclusions . . . . .	48
9.3	Future work. . . . .	50
	<b>Bibliography</b>	<b>52</b>
<b>A</b>	<b>CSV Test Result Files</b>	<b>55</b>
A.1	Context for each CSV file . . . . .	55
A.2	CSV file structure . . . . .	56
<b>B</b>	<b>Jupyter Notebooks</b>	<b>57</b>
B.1	Notebook structure: Netcat - dd_udp.c . . . . .	57
<b>C</b>	<b>Data transfer calculations</b>	<b>61</b>
<b>D</b>	<b>SndbufErrors &gt; 0 measurements</b>	<b>63</b>

# LIST OF FIGURES

1.1	Depiction of a data diode supplying confidentiality protection to a higher classified network [Stevens, 1999]. . . . .	2
2.1	Internet Model with basic dataflow. . . . .	4
2.2	Linux on the Internet Model and the encapsulation of UDP data (derived from [Maxnilz, 2025]). . . . .	5
2.3	The UDP header [Bhalodia, 2021]. . . . .	6
2.4	A received UDP Jumboframe sent with Netcat reassembled from Ethernet frames. . . . .	8
2.5	Basic data diode architecture with TCP proxies (adopted from [Kim and Min, 2016]). . . . .	10
3.1	Setup used by Honggang [Honggang, 2013]. . . . .	12
3.2	Schema of the OSDD setup from Vrolijk [Vrolijk, 2023] as implemented by Story [Story, 2023b]. . . . .	13
3.3	Mediaconverter as used by Story [Story, 2023b]. The RX/TX fiber optic ports are positioned on the left, the RJ45 connection on the right. . . . .	14
3.4	Example of the chunk-based mechanism as implemented by Story's pydiode [Story, 2023b]. . . . .	14
4.1	Data diode test setup, implements Figure 3.2. . . . .	17
4.2	The GMKtec NucBox M5 plus. . . . .	18
4.3	System boundary of the test setup. . . . .	21
6.1	Mapping the relevant measuring points on the Internet Model (derived from [Maxnilz, 2025]). . . . .	30
7.1	Fragmentation of UDP datagrams sent by Netcat with MTU=9194 bytes. . . . .	40
7.2	Fragmentation of UDP datagrams sent by pydiode with redundancy=1 and MTU=9194 bytes. . . . .	40
7.3	Problem with UDP datagrams sent by Netcat; MTU=9194 bytes; segment size of 9166 bytes. . . . .	42
8.1	Basic data diode architecture with TCP proxies (adopted from [Kim and Min, 2016]). . . . .	47
A.1	Example of CSV file directory stored on GitHub . . . . .	55

# LIST OF TABLES

5.1	Initial Netcat test results with default UDP.c and dd_udp.c kernel modules. . .	26
5.2	Initial pydiode test results with default UDP.c and dd_udp.c kernel modules. .	26
7.1	Adjusted receiver socket buffers: Netcat test results. . . . .	36
7.2	Adjusted receiver socket buffers: pydiode test results with redundancy=1. . .	36
7.3	Adjusted receiver socket buffers: pydiode test results with redundancy=2. . .	37
7.4	Adjusted receiver socket buffers and MTU=9194: Netcat test results. . . . .	38
7.5	Adjusted receiver socket buffers and MTU=9194: pydiode test results with re- dundancy=1. . . . .	39
7.6	Adjusted receiver socket buffers and MTU=9194: pydiode test results with re- dundancy=2. . . . .	39
7.7	Adjusted rcv socket buffers; MTU=9194: pydiode with redundancy=1; 9166 byte data segments. . . . .	43
7.8	Adjusted rcv socket buffers; MTU=9194: pydiode with redundancy=2; 9166 byte data segments. . . . .	43
8.1	Validating baseline Netcat test results. . . . .	45
8.2	Validating baseline pydiode test results. . . . .	45
A.1	Fields within the CSV files. . . . .	56
D.1	Average increase in SndbufErrors values per file transferred; pydiode redun- dancy=1, UDP rcv socket buffer 4MiB. . . . .	63
D.2	Number of file transfers where SndbufErrors > 0; pydiode redundancy=1, UDP rcv socket buffer 4MiB. . . . .	63



# 1

## INTRODUCTION

In a world where both cybercrime and cyber operations by national actors are continuously increasing [NCTV, 2023], the importance of protecting the crown jewels of organizations is growing ever more significant.

Attacks on corporate networks can originate both externally or internally within a company. A widely accepted tactic to counter both types of attacks is network segmentation [Cisco]. Network segmentation is the practice of dividing a computer network into smaller, isolated subnetworks to improve security, performance, and manageability. Network segments can be implemented both logically and physically. In the case of physically separated networks, these networks are also referred to as airgapped networks.

By segmenting networks, it is possible to create networks with different classification levels. When dealing with different classification levels, it is important that dataflows between these network segments occur through well-implemented interfaces.

With strictly airgapped networks, this presents an additional challenge, as these networks are physically separated. Traditionally, data carriers (such as USB sticks) and procedures (i.e. malware scanning and data integrity checks) are used. These methods help facilitate secure data transfer from lower to higher classified networks but also present challenges in terms of usability and the risks of human error [Arnold, 2016].

In addition, modern solutions such as data diodes are available. A physical data diode is a hardware device that enforces one-way communication by allowing data to flow in only one direction at the physical layer [Stevens, 1999], making reverse transmission physically impossible. A virtual data diode achieves the same unidirectional control in software or virtualized environments [de Freitas et al., 2018]. Virtual data diodes offer flexibility and lower cost, but they rely on software integrity. Physical data diodes provide the strongest security guarantees, offering a clear advantage in that regard [Story, 2023b].

The enforcement of unidirectional data traffic makes data diodes useful for critical infrastructures where safety is paramount [Arnold, 2016]. Due to the unidirectional data traffic, no data is leaked from the receiving environment towards the sending environment, something that is possible with bidirectional data traffic.

### 1.1. MOTIVATION

The importance of unidirectional data streams is increasing rapidly. More and more sensors and data sources are providing increasingly detailed data, resulting in larger and more

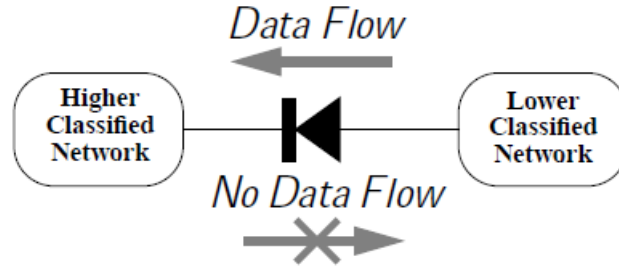


Figure 1.1: Depiction of a data diode supplying confidentiality protection to a higher classified network [Stevens, 1999].

extensive data streams. These data sources can be either trusted or untrusted. When transmitting these resulting data streams to higher-classified networks, ensuring secure transfer is essential.

Certified physical data diodes are relatively expensive, with the lowest-priced option estimated at €15,000 [Delta, 2019]. A wide range of products is offered by various suppliers.

One of the biggest issues with data diodes is the phenomenon of packet loss. Due to the unidirectional data traffic, using a protocol such as UDP, the receiving side cannot communicate with the sender about the status of the data transmission. This can lead to data loss both in the event of issues within the physical layer and in case of bufferoverflow on the receiving side if the data stream turns out to be larger than expected [Honggang, 2013]. Packet loss poses a threat to the integrity of the data to be transmitted through a data diode.

Academic publications on data diodes remain relatively scarce. Therefore, this thesis aims to contribute to the following areas:

- research on mitigating packet loss,
- implementing and testing measures to eliminate packet loss,
- so that cheaper and more reliable performing data diodes become available to our society and
- a useful addition to existing academic publications on data diodes.

## 1.2. RESEARCH QUESTIONS

Since this research requires access to the source code of both the operating system and the system software responsible for handling UDP traffic, Linux was selected. One may assume the Linux UDP.c kernel module to be a mature piece of software, fit for the majority of use cases where UDP data transfer is needed. However, it will not be optimized for use with(in) data diodes specifically. Thus this module will not be implemented with minimizing packet loss in data diodes in mind. It is reasonable to assume that compromises have been made in favor of higher data transmission rates over preventing packet loss. These considerations lead to the following research questions:

1. How can the Linux UDP.c kernel module be implemented or modified with minimizing packet loss during data transfer in mind?
2. What are relevant measuring points for monitoring UDP data transfer, to facilitate

enhancements aimed to minimizing packet loss?

3. What is the impact of adjusting the UDP receiver socket buffer size on packet loss during transmission?
4. The Maximum Transmission Unit (MTU) defines the largest packet size a network interface can handle. How does changing the MTU to its maximum value influences packet loss during the transmission of UDP datagrams?
5. When applications transmit data, it is usually divided into segments before being sent. This raises the question of how aligning data segment sizes with the maximum MTU value affects transmission performance and packet loss.

### 1.3. OVERVIEW

Chapter 2 describes relevant background information. It covers UDP and the Internet Model. It also provides insight into Jumbo Frames and MTU. The basic architecture of data diodes is discussed. Finally, attention is given to methods of preventing packet loss and the calculation of loss ratios.

Chapter 3 discusses related work. It outlines the hardware setup of the diode and the results obtained with it in 2023 during previous research [Story, 2023b], which form the foundation for this research. It also describes MTU and related efficiency ratios. Addressing the research questions is described in Chapter 4. It also describes the test setup, the tests to be conducted, and the required scripts.

Chapter 5 describes the functionality of the standard UDPc Linux kernel module. In addition, an alternative UDP module is introduced. The chapter concludes with a comparative test. The results of the investigation into the measurement points from research question 2 are presented in Chapter 6. The results are mapped onto the Internet Model, and the chapter concludes with several observations. The results of the research addressing research questions 3, 4, and 5 are presented in Chapter 7.

Chapter 8 discusses issues concerning the validity and reliability of the obtained results. Chapter 9 contains the conclusions of this research. In addition, suggestions for future work are presented.

# 2

## BACKGROUND

This chapter discusses topics related to the User Datagram Protocol (UDP), the basic architecture of data diodes, and known methods to reduce packet loss. To limit scope of this research, the use of Internet Protocol version 4 (IPv4) and Ethernet II framing is assumed.

### 2.1. OVERVIEW OF UDP AND THE INTERNET MODEL

The Internet Model -shown in Figure 2.1- organizes network communication into four layers: Application, Transport, Internet, and Link. As a connectionless protocol, UDP follows a straightforward path through these layers.

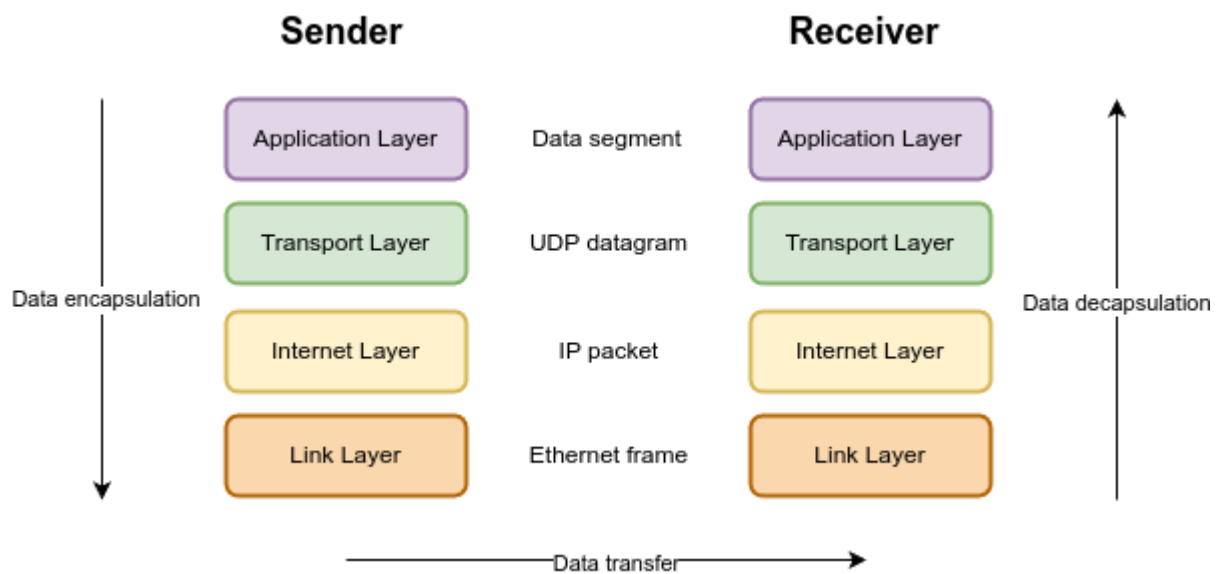


Figure 2.1: Internet Model with basic dataflow.

#### SEGMENTS, DATAGRAMS, PACKETS AND FRAMES

The terminology for the data units are not always clearly defined in publications. In this thesis, *(Data) segments* refer to data at the Application Layer, *(UDP) datagrams* are units of data at the Transport Layer, and *(IP) packets* refer to data at the Internet Layer. Finally, *(Ethernet) frames* are the units sent and received by the Link Layer.

### 2.1.1. UDP DATAFLOW THROUGH THE INTERNET MODEL

Below follows a description of the flow for both the sending and receiving sides on Linux systems. This is a high level description, but it contains technical terms. These technical details will be discussed later on in this section. Please refer to Figure 2.2 when headers are mentioned.

#### SENDER

In the *Application Layer*, the sending application (e.g., Netcat or pydiode) creates a UDP socket and sends data to it as a segment using a system call. The kernel's UDP stack -which is positioned at the *Transport Layer*- takes the data, adds a UDP header, and creates a UDP datagram. For large data, a segment might be split into multiple datagrams, depending on the the size of the data send by the application. The *Internet Layer* adds an IP header. If the datagram exceeds the network's Maximum Transmission Unit (MTU), it is fragmented into smaller IP packets. Within the *Link Layer*, each IP packet is encapsulated in an Ethernet frame by adding an Ethernet header and footer (containing a 32-bit checksum) to the packet. The frame is then queued in the Network Interface Card (NIC) transmit buffer, from which it is sent through the data diode.

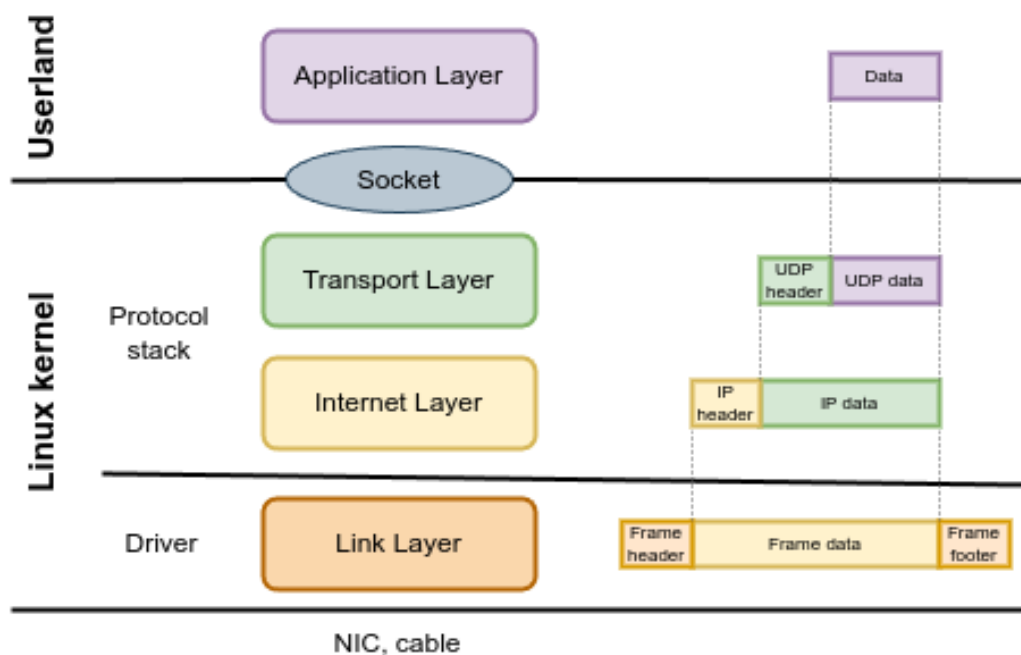


Figure 2.2: Linux on the Internet Model and the encapsulation of UDP data (derived from [Maxnilz, 2025]).

#### NETWORK TRANSMISSION

The frames travel through the the diode hardware. Losses or corruption can happen due to hardware issues or filtering. These losses can not be measured directly; they can only be inferred by comparing sender and receiver metrics.

#### RECEIVER

The receiver's NIC -positioned in the *Link Layer*- receives the frame. The Ethernet checksum of the frame is calculated. If the checksum matches the checksum within the Ethernet

footer, the Ethernet header and footer are stripped and the IP packet is passed to the kernel. Otherwise the frame is dropped. This process is required by the Ethernet standard. If the NIC's receive buffer is full, packets may be dropped also.

In the *Internet Layer*, the kernel processes the IP packet. It verifies the IP header and reassembles fragments if the original datagram was fragmented. Packets with invalid headers are discarded at this stage. At the *Transport Layer*, the UDP stack processes the datagram. The UDP checksum is verified (if enabled), and the datagram is delivered to the corresponding UDP socket buffer. If no application is listening on the destination port, or if the socket buffer is full, the datagram is dropped. Returning to the *Application Layer*, the receiving application (e.g., Netcat) reads the data from its dedicated UDP socket buffer for processing or storage.

## 2.2. THE UDP PROTOCOL

UDP is a connectionless protocol, based on the RFC 768 specification [Postel, 1980]. This specification describes the goal of UDP as

*"a procedure for application programs to send messages to other programs with a minimum of protocol mechanism ... and delivery and duplicate protection are not guaranteed."*

As a consequence of the 'minimum of protocol mechanism', UDP does not require acknowledgements to operate reliably. This is especially valuable in the context of a data diode, where acknowledgements are not feasible due to the enforced unidirectional nature of the connection. This makes UDP a fitting protocol for use with data diodes.

### 2.2.1. UDP HEADER

The UDP header is a simple and lightweight structure that provides essential metadata for transmitting data packets. It consists of four 16 bit long fields, resulting in a total header size of 8 bytes.

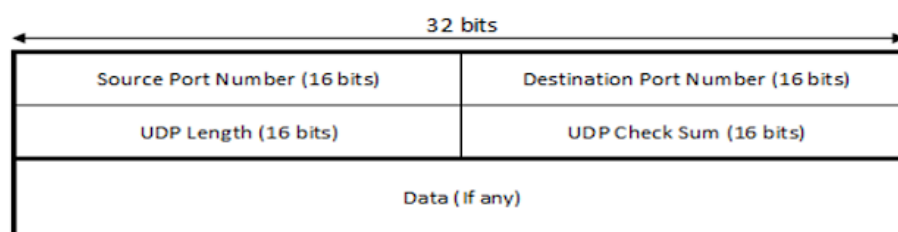


Figure 2.3: The UDP header [Bhalodia, 2021].

The optional 16-bit *Source Port Number* field identifies the port number of the sending application. If not used, it is set to zero. The 16-bit *Destination Port Number* specifies the port number of the receiving application on the destination device. It ensures the packet is delivered to the correct application. With 16 bits, there are 65,536 possible port numbers, though ports below 1024 are usually reserved for well-known services. The total length of the UDP packet is indicated by the 16-bit *UDP Length* field, including the 8-byte header and the data that follows. The minimum value is 8 bytes (header only, no data), and the theoretical maximum is 65,535 bytes, though limits like the network's Maximum Transmission Unit (MTU) often reduce this. Finally, the optional 16-bit *UDP Checksum* might be used for error-checking. When not used, this checksum is set to zero.

### 2.2.2. UDP CHECKSUM

When UDP operates over IPv4, the checksum mechanism is defined as an optional feature. It helps to ensure the integrity of the transmitted data by detecting errors that might occur during transmission.

#### HOW THE UDP CHECKSUM IS CALCULATED

The UDP checksum is a 16-bit one's complement sum computed over the *UDP header*, the *data payload* and a *pseudo-header* [Postel, 1980], following this process:

1. Create the pseudo-header.  
The pseudo-header is a 12-byte sized structure that is not transmitted, but is constructed for checksum calculation only. This header includes:
  - Source IP address: 4 bytes.
  - Destination IP address: 4 bytes.
  - Protocol: 1 byte (set to 17 for UDP).
  - UDP length: 2 bytes (the total length of the UDP header and the data).
  - Zero padding: 1 byte.
2. Assemble the data.  
Combine the pseudo-header, the UDP header (with the checksum field initially set to 0) and the data payload. If the total length is odd, append a 0 byte to make it an even number of bytes.
3. Sum the 16-bit words and handle overflow.  
If the sum exceeds 16 bits, fold the extra bits back into the lower 16 bits by adding them to the sum.
4. Take the One's Complement.
5. Insert the checksum into the UDP header.

The optional UDP checksumming is a simple way to detect errors in transmitted data. But it is based on 16-bit values regardless of the size of the datagrams. UDP checksumming will not detect errors if the resulting checksum remains the same. When datagram sizes increase, these collisions are more likely to occur.

UDP datagrams are encapsulated in Ethernet frames before data transmission. Ethernet frames contain footers with a 32-bit CRC [Rijsinghani, 1994]. Due to their size, these CRC's will be less liable to collisions. These Ethernet frame checksums are mandatory. When a NIC receives a frame with a non matching checksum, the frame will be rejected. This means that erroneous frames are expected to be discarded by the receiving NIC before the UDP checksum is evaluated (if it is enabled at all).

## 2.3. DATA TRANSFER WITH JUMBOFRAMES

### 2.3.1. MTU

The MTU (Maximum Transmission Unit) is the largest packet size a network interface can handle. Larger packets will be split into smaller fragments. Unless specified explicitly, network interfaces use a default MTU of 1500 bytes, which is the standard for Ethernet networks [Murray et al., 2012]. The MTU includes the *IP header* (typically 20 bytes for IPv4), the *transport protocol header* (8 bytes for UDP), and the *payload* (the actual data being sent).

For example: when dealing with UDP over IPv4, the IP header takes 20 bytes and the UDP header 8 bytes. Assuming the default MTU of 1500 bytes, this results in a maximum



payload of  $1500 - 20 - 8 = 1472$  bytes.

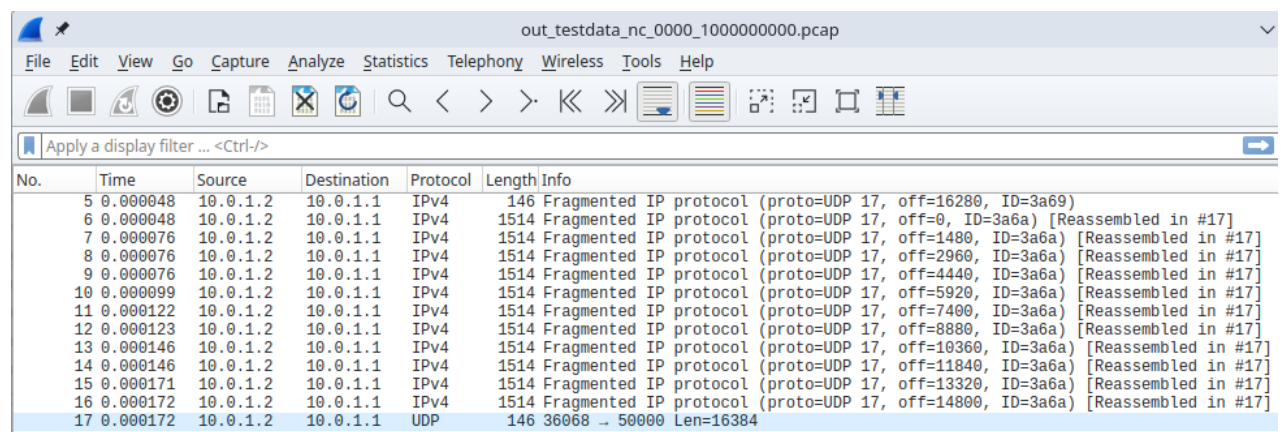
The maximum MTU value is bound to the IPv4 *Total Length* header field, which is 16 bits. Hence the maximum MTU value is 65,535 bytes. In that case the maximum payload, for UDP over IPv4, becomes  $65,535 - 20 - 8 = 65,507$  bytes. But this is a theoretical value, which may not be supported by the hardware and/or by Linux device drivers.

### 2.3.2. JUMBOFRAMES

Jumboframes are Ethernet frames with a payload that exceeds the default MTU of 1500 bytes [Wright and Scarpati, 2023]. They often reach sizes of up to 9000 bytes or more, depending on the network hardware and configuration. Ethernet frames consists of:

- A 14 byte Ethernet header which contains the MAC addresses of the sender and the receiver. It also contains an EtherType determining the kind of encapsulated packet.
- The encapsulated IP packet.
- A 4 byte Ethernet footer that holds a 32-bit CRC.

### 2.3.3. MTU AND THE HANDLING OF JUMBOFRAMES



No.	Time	Source	Destination	Protocol	Length	Info
5	0.000048	10.0.1.2	10.0.1.1	IPv4	146	Fragmented IP protocol (proto=UDP 17, off=16280, ID=3a69)
6	0.000048	10.0.1.2	10.0.1.1	IPv4	1514	Fragmented IP protocol (proto=UDP 17, off=0, ID=3a6a) [Reassembled in #17]
7	0.000076	10.0.1.2	10.0.1.1	IPv4	1514	Fragmented IP protocol (proto=UDP 17, off=1480, ID=3a6a) [Reassembled in #17]
8	0.000076	10.0.1.2	10.0.1.1	IPv4	1514	Fragmented IP protocol (proto=UDP 17, off=2960, ID=3a6a) [Reassembled in #17]
9	0.000076	10.0.1.2	10.0.1.1	IPv4	1514	Fragmented IP protocol (proto=UDP 17, off=4440, ID=3a6a) [Reassembled in #17]
10	0.000099	10.0.1.2	10.0.1.1	IPv4	1514	Fragmented IP protocol (proto=UDP 17, off=5920, ID=3a6a) [Reassembled in #17]
11	0.000122	10.0.1.2	10.0.1.1	IPv4	1514	Fragmented IP protocol (proto=UDP 17, off=7400, ID=3a6a) [Reassembled in #17]
12	0.000123	10.0.1.2	10.0.1.1	IPv4	1514	Fragmented IP protocol (proto=UDP 17, off=8880, ID=3a6a) [Reassembled in #17]
13	0.000146	10.0.1.2	10.0.1.1	IPv4	1514	Fragmented IP protocol (proto=UDP 17, off=10360, ID=3a6a) [Reassembled in #17]
14	0.000146	10.0.1.2	10.0.1.1	IPv4	1514	Fragmented IP protocol (proto=UDP 17, off=11840, ID=3a6a) [Reassembled in #17]
15	0.000171	10.0.1.2	10.0.1.1	IPv4	1514	Fragmented IP protocol (proto=UDP 17, off=13320, ID=3a6a) [Reassembled in #17]
16	0.000172	10.0.1.2	10.0.1.1	IPv4	1514	Fragmented IP protocol (proto=UDP 17, off=14800, ID=3a6a) [Reassembled in #17]
17	0.000172	10.0.1.2	10.0.1.1	UDP	146	36068 → 50000 Len=16384

Figure 2.4: A received UDP Jumboframe sent with Netcat reassembled from Ethernet frames.

When applications try to send data leveraging Jumboframes, while the NICs are configured with the default MTU value of 1500 bytes, the Jumboframes will be broken into multiple frames. As an example, let us examine in detail what happens when Netcat is offered a 16384 byte chunk to send using UDP, while the NICs default MTU values have not been modified.

#### WHAT NETCAT DOES

When Netcat receives the 16384 byte chunk, it aims to send it as a single data segment to the socket buffer. In the UDP context within the kernel, this becomes a UDP datagram. The UDP datagram consists of the UDP Header (8 bytes) and the data payload of 16384 bytes. This results in total UDP datagram size of  $8 + 16384 = 16392$  bytes.

This 16392 byte datagram is passed to the Internet Layer, assuming it can be sent as a single Jumboframe of 16384 bytes (plus headers). However, the NICs' MTU setting will override this assumption.



### INTERNET LAYER AND MTU MISMATCH

Since the Internet Layer is constrained by the 1500 byte MTU, the 16392 bytes of the UDP datagram have to be fragmented into multiple IP packets. This is shown in Figure 2.4.

The UDP datagram is split into:

- 1 packet of 1500 bytes, which contains a 20 byte IP header and 1480 byte payload, of which the first 8 bytes are the UDP header.
- 10 packets of 1500 bytes, containing a 20 byte IP header and 1480 byte payload each, consisting of actual data only.
- 1 packet of 132 bytes, with a 20 byte IP header and 112 byte payload, containing actual data only.

It should be noted that when fragmentation occurs under IPv4, the size of the payloads of the packets mentioned in the first two bullets must be a multiple of 8 bytes. This is required by the Internet Protocol specification (RFC 791 [Postel, 1981]). The 1480-byte payloads meet this requirement.

### TRANSMITTING AND RECEIVING

The IP packets will be transmitted over the network. But before the NIC sends the packets, the Ethernet header (14 bytes) is added. Thus converting each IP packet into an Ethernet frame. This causes 11 Ethernet frames of  $1500 + 14 = 1514$  bytes and 1 Ethernet frame of  $132 + 14 = 146$  bytes to be send.

This is in accordance with Figure 2.4. However, network sniffing tools are not able to show the 4 byte Ethernet footers. These footers are added and removed by the NIC's just before network sniffing tools have access to the frames.

At the receiving side, the Ethernet headers and footers are removed and the receiver's Internet Layer reassembles the IP packets into the 16392 byte UDP datagram, delivering 16384 bytes to the application, assuming no fragments are lost.

## 2.4. BASIC ARCHITECTURE OF DATA DIODES

Figure 2.5 shows the basic architecture of a functional data diode. The box labeled 'Unidirectional Transmission System' is the actual data diode. It consists of two TCP proxies with a unidirectional connection. This connection is usually created using fiber optics, with only the light source on the transmitting side activated, physically separating the electric circuitries of both proxies [An and Tin, 2021; Honggang, 2013; Kim and Min, 2016; Stevens, 1999; Story, 2023b; Vrolijk, 2023].

Network traffic typically uses the bidirectional TCP. When the *TCP Sender* wants to send data to the *TCP Destination*, it connects to the *TCP Sending Proxy*. This proxy converts the network traffic to a unidirectional datastream (usually using UDP) and sends it to the *TCP Receiving Proxy*. The latter proxy sends the received data using TCP to the *TCP Destination*. For the transition from TCP to UDP and vice versa, both proxies need buffers to perform these conversions correctly.

## 2.5. PACKET LOSS

Although the term packet loss has already been introduced earlier in this study, its precise definition has not yet been established.

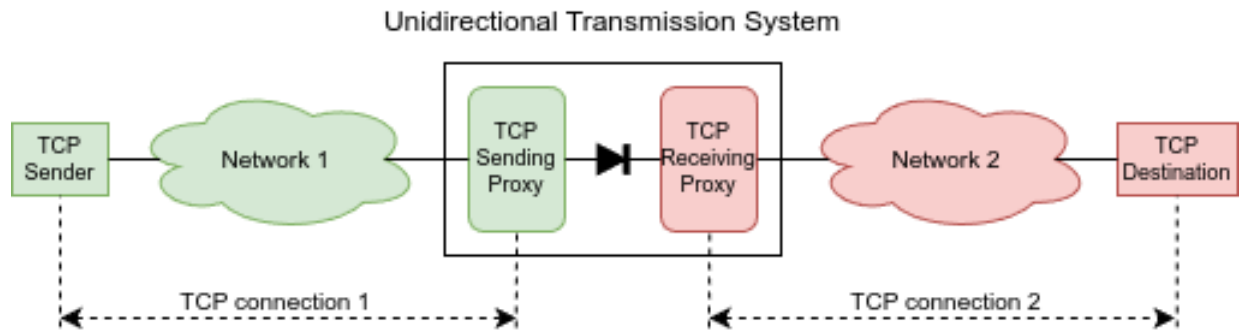


Figure 2.5: Basic data diode architecture with TCP proxies (adopted from [Kim and Min, 2016]).

According to the IETF IP Performance Metrics (IPPM) framework (RFC 7680) [Almes et al., 2016], packet loss is formally defined at the Internet Layer as the failure of an IP packet transmitted at the sender to arrive at the receiving measurement point within a specified time interval, relative to the total number of packets sent.

In practice, packet loss may also be observed at higher protocol layers. At the Transport or Application Layer, loss can occur when UDP datagrams fail to reach the receiving socket. Thus, while the IETF provides a precise, Internet Layer definition of packet loss, the phenomenon may manifest differently depending on whether it is measured at the Internet Layer, the Transport Layer, or the Application Layer.

## 2.6. KNOWN METHODS TO REDUCE PACKET LOSS

Due to UDP being a unidirectional data transfer protocol, the receiving side cannot communicate with the sender about the status of the data transmission. This can lead to data loss both in the event of issues within the Link Layer and in case of bufferoverflows on the sending side as well as on the receiving side, if the data stream turns out to be larger than expected [Honggang, 2013].

*Bitrate limiting* [Khan et al., 2018] is the practice of controlling the rate at which data is transmitted across the data diode. In a UDP-based data diode, the sending proxy cannot adjust its sending rate based on network conditions, because there are no responses from the receiving proxy. Without limitation, the sender can overwhelm the receiver by saturate its buffers, leading to packet loss and degraded performance. By limiting the bitrate, one can effectively align the sender's data rate with the receiver's processing capacity and buffer constraints. This makes the data flow more stable and manageable. However, this may introduce additional processing overhead in the sending proxy, since all data must be buffered before transmission.

*Redundant transmissions* involves the intentional duplication of UDP datagrams to compensate for the potential loss of data [Honggang, 2013]. In a data diode architecture, re-transmissions are not feasible due to its unidirectional nature. For this reason, redundancy is introduced at the sending proxy to mitigate the effects of packet loss caused by buffer overflow or hardware problems. This approach increases bandwidth utilization, but it significantly increases the likelihood that all data arrives at the receiving proxy intact. This provides an effective countermeasure against data loss in data diodes where reliability is a priority.

Proper *adjustment of buffer sizes* at both the sender and receiver proxies is a consideration in data diode design. Inadequate buffer sizes can lead to packet loss, especially during traffic bursts [Honggang, 2013]. However, excessively large buffers may introduce unwanted delays due to the additional processing required to manage them. By tuning the buffer sizes appropriately, system designers can balance throughput, latency, and reliability constraints. This process is especially significant in high-throughput environments, where buffer behavior can directly affect the effectiveness of the data diode.

*Forward Error Correction (FEC)* is a method for achieving higher reliability in data diode communication by embedding redundant information within the transmitted data [Rizzo, 1998]. FEC techniques enable the receiving side to reconstruct lost or corrupted packets, negating the need for retransmission requests. FEC is especially suited for unidirectional UDP transfers, where backward error recovery is not feasible. By leveraging such error-correcting codes, data diode systems can tolerate higher levels of packet loss and maintain data integrity.

In this research, bitrate limiting and redundant transmissions are applied. This functionality is supported by the tools described in Chapters 3 and 4. Buffer size adjustments are also applied, which are further explained in Chapter 7.

## 2.7. LOSS RATIOS

Honggang introduced a few simple loss-related metrics [Honggang, 2013]. Two of these metrics are cited below:

The first metric is the *Packet Loss Ratio (PLR)*:

$$\text{PLR [\%]} = \frac{\text{LostPackets}}{\text{SendPackets}} \times 100\% \quad (2.1)$$

In cases where lost packets are not easily measured, the *Completed File Ratio (CFR)* can be used as an alternative metric to assess the effectiveness of a data diode:

$$\text{CFR [\%]} = \frac{\text{CompletedFiles}}{\text{SendFiles}} \times 100\% \quad (2.2)$$

# 3

## RELATED WORK

Data diodes can be relatively inexpensive to build (less than €100) as shown by the Open-Source Data Diode (OSDD) project from Vrolijk [Vrolijk, 2023]. However, the price of the cheapest certified data diode available is estimated at €15,000 [Delta, 2019]. Between these extremes, there is a wide range of products available from various suppliers. Kerkhof [Kerkhof, 2021] provides an overview of state-of-the-art data diodes in 2021. Most products listed support data throughput rates of at least 1 Gbps. Some products even promise a throughput of up to 100 Gbps.

### 3.1. UNIDIRECTIONAL DATATRANSFER AND PACKET LOSS

In 2013, Honggang [Honggang, 2013] attributed packet loss to issues in the Link Layer (link or equipment error) and to buffer overflow (congestion packet loss).

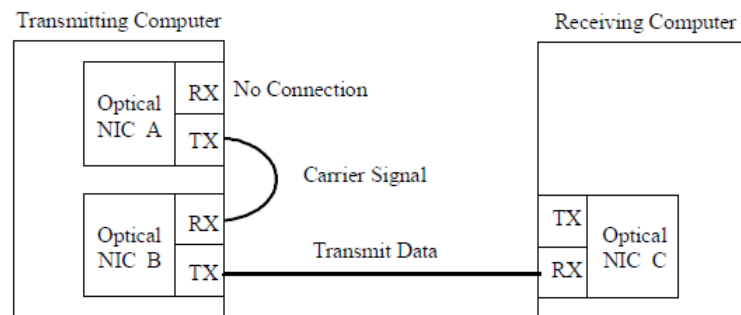


Figure 3.1: Setup used by Honggang [Honggang, 2013].

During experiments using the simple test setup from Figure 3.1, Honggang measured a Packet Loss Ratio of 0,0 % using UDP with data transmission rates up to 300 Mbps and a packet size of 1460 bytes. These results were achieved without any additional measures like FEC (Forward Error Control).

Honggang states packet loss always is *"random packet loss due to link transmission errors"* or *"congestion packet loss"*. While this seems to be a reasonable assumption, no proof of that statement is given.

Kim and Min [Kim and Min, 2016] proposed the RED (REliable Data diode) in 2016. RED is based on both limiting the transmissionrate by delaying transmission of TCP pack-

ets from the sender and the transmission of duplicate packets to the receiver. There is no known implementation of RED, so no test results are available.

In 2021, An and Tin [An and Tin, 2021] proposed the V10-DATADIODE. Their experiment showed no packet loss at transmission rates lower than 300 Mbps. The V10-DATADIODE utilizes data sequencing and FEC. It also offers web application interfaces to assist users in both sending and receiving files.

### 3.2. A PRACTICAL APPROACH

In his 2023 article, Story [Story, 2023b] uses the OSDD setup from Vrolijk [Vrolijk, 2023] for his experiments. The main reasons for this choice are the low cost of the components, the easy configuration and the reported transfer speed of 600 Mbps.

#### 3.2.1. THE VROLIJK SETUP

Vrolijk offers a number of low cost data diode setups. The one used by Story is called 'Secure basic setup using mediaconverters and Y-cable'. This setup, as shown in Figure 3.2, connects the sender and receiver using gigabit ethernet fiber media converters. These converters are used to convert data from ethernet (copper) to fiber optics and vice versa. The media converter used during this research is shown in Figure 3.3. The setup also needs a fiber splitter. One of the splitter's outputs needs to connect to the receive (RX) port of the send converter, and the other output to connect to the RX port of the receive converter. Without this additional loop, Ethernet autonegotiation will fail and data transfer will not commence [Story, 2023b].

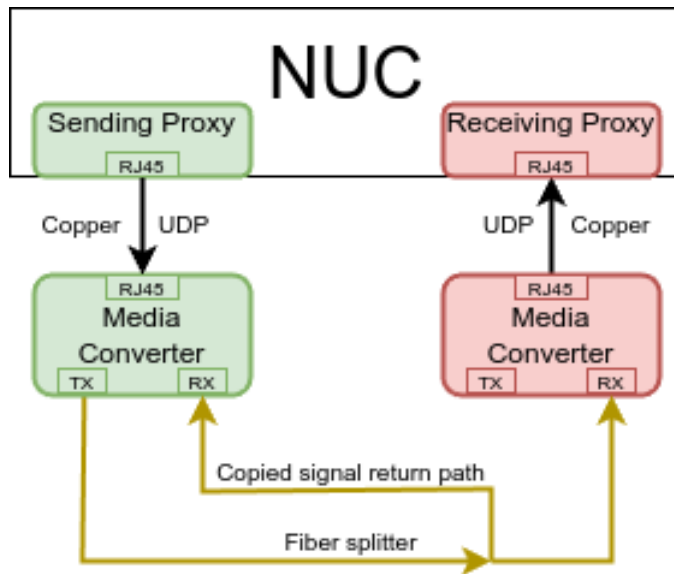


Figure 3.2: Schema of the OSDD setup from Vrolijk [Vrolijk, 2023] as implemented by Story [Story, 2023b].

Story uses a NUC (Next Unit of Computing) with two distinct network interface cards (NIC's) to send and receive data through the OSDD setup. This means that his NUC functions as both the TCP Sending Proxy and the TCP Receiving Proxy shown in Figures 2.5 and 3.2. This makes it easier to measure performance and packet loss. Obviously, this is not the way data diodes are meant to be used. Nonetheless, Figure 3.2 shows an implementation of the 'Unidirectional Transmission System' from Figure 2.5.



Figure 3.3: Mediaconverter as used by Story [Story, 2023b]. The RX/TX fiber optic ports are positioned on the left, the RJ45 connection on the right.

During the experiments, the setup is tested using the open-source data transfer utilities Netcat [NMAPORG, 2024], UDPcast [UdpCast, 2016] and pydiode [Story, 2023a] to send and to receive data. This software runs on the NUC, allowing to connect both sides of the data diode without the need for extra components.

Story’s results show Netcat being outperformed by both UDPcast and pydiode. While UDPcast is configured to use FEC and rate limiting, it was outperformed by pydiode. Pydiode achieves a 0,0% packet loss for sending 1 Gbit of data with a bitrate of 1 Gbps, with a redundancy factor of 2, within an average time of 2,17 seconds.

### 3.2.2. PYDIODE: CHUNKS, RATE LIMITING AND REDUNDANT TRANSMISSION

Pydiode’s main characteristic is the use of a simple scheme for sending packets in chunks, with the option of redundant transmission of these chunks. Users can also specify a maximum bitrate. An example of this chunk-based mechanism is shown in Figure 3.4.

	Red Chunk						Blue Chunk						Red Chunk						Black Chunk			
Sequence Number	0	1	2	0	1	2	0	1	2	0	1	2	0	1	2	0	1	2	0	0	0	0
Payload	A	B	C	A	B	C	D	E	F	D	E	F	G	H	I	G	H	I				

Figure 3.4: Example of the chunk-based mechanism as implemented by Story’s pydiode [Story, 2023b].

In this example, packets A, B and C make up the first red chunk which is sent twice. The blue chunk consists of packets D, E and F, and is sent twice. Packets G, H and I are in the last red chunk which is also sent two times. The transmission ends with a black chunk. A pydiode packet consists of a header and a payload. The header takes up 5 bytes containing the chunk color, the number of packets in the chunk, and the sequence number of the packet.

Pydiode is implemented in Python and uses the standard Python *asyncio* library [PSF, 2025], which is also implemented in Python. The *asyncio* library enables concurrency for I/O-bound programs by providing a single-threaded event loop and high-level primitives to structure asynchronous code using the `async/await`<sup>1</sup> syntax.

<sup>1</sup><https://docs.python.org/3/library/asyncio-task.html>

The fact that pydiode outperforms UPDCAST is remarkable since UPDCAST is mainly implemented in C, which runs faster than Python. Of course, asyncio must use OS system calls which will be written in C. But it is still an achievement to outperform other specialized data transfer tools using only Python, a simple chunk-based mechanism, redundant transmission, bit rate limiting, and an asynchronous I/O concurrency library.

### 3.3. MTU AND EFFICIENCY

As discussed in Section 2.3, increasing the MTU beyond the default value of 1500 bytes enables the use of Jumbo Frames for data transfer. In their 2012 paper, [Murray et al., 2012] observe that the default MTU value has been 1500 bytes since its standardization in 1982. While this paper primarily focuses on TCP/IP, it also highlights several points that are relevant to UDP traffic in data diodes.

According to Murray et al., leveraging Jumboframes leads to *lower overhead*. With fewer packets, there is less metadata (headers and footers) to process, making better use of available bandwidth. A *reduced CPU Load* is expected because network devices and servers handle fewer packets, decreasing the processing demand on CPUs. Finally a *higher throughput* will result from carrying more payload per frame.

In Section 2.3.1, it is established that with a standard MTU of 1500 bytes, the data payload (excluding the UDP and IP headers, which together account for 28 bytes) is 1472 bytes. The resulting Ethernet frame is 18 bytes larger than the MTU due to the addition of the Ethernet header and trailer and becomes 1518 bytes. The data efficiency per transmitted frame is therefore given by

$$\eta_{mtu=1500} = \frac{1472}{1518} \times 100\% = 96.97\%.$$

For a Jumboframe of 9000 bytes, the MTU becomes  $9000 - 18 = 8982$  bytes (as inferred by Equation C.3) and the maximum data payload is  $8982 - 28 = 8954$  bytes. In that case, the data efficiency is

$$\eta_{mtu=9000} = \frac{8954}{9000} \times 100\% = 99.48\%.$$

Thus, changing the MTU from 1500 to 8954 bytes results in a 2.5 % increase in data efficiency per Ethernet frame.

Several limitations are also recognized. For Jumboframes to work effectively, all devices in the network path must support them. Otherwise, it could lead to issues like packet fragmentation or loss, which would undermine the advantages [Wright and Scarpati, 2023].

Given the significant difference between the maximum and the default MTU sizes (65,535 vs. 1500 bytes), the reliability of the Ethernet CRC-32 could be at risk. [Murray et al., 2012] state that the use of 9000-byte MTU's in high performance LAN's shows the effectiveness of the CRC-32. However, notably larger frame sizes will need a more robust checksum mechanism [Jain, 1990].



# 4

## RESEARCH

### 4.1. METHOD

The methodology used to answer the research questions posed in Chapter 1 can be described in the following steps.

#### RESEARCH QUESTION 1

This study begins with a review of background information concerning Linux kernel modules, followed by an analysis of the functionality provided by the default UDP.c module. The research then identifies the minimal set of features required in a UDP.c kernel module to facilitate the transmission and reception of data within the context of a data diode. Building on this analysis, an alternative UDP kernel module is implemented in the C programming language. Both the original and the modified modules are subsequently subjected to testing, and the resulting performance data are compared and analyzed.

#### RESEARCH QUESTION 2

The research begins with a review of background information concerning the Linux network stack, followed by an examination of potential measurement points. From these, suitable measurement points are selected, and measurement functionality is implemented using them. Finally, tests are performed leveraging the selected measurement points.

#### RESEARCH QUESTIONS 3, 4 AND 5

To answer these research questions, the data diode proxies are modified accordingly, followed by a series of tests. These tests leverage the measurement points from the previous question. The proxy settings adjusted in this study include the UDP receiver buffer size, the MTU, which is set to its maximum supported value, and the data segment sizes, which are aligned with the maximum MTU. The test results are then analyzed and compared.

### 4.2. TESTS AND EXPERIMENTS

The tests and the experiments are conducted similar to the experiments by Story [Story, 2023b], which are based on the OSDD setup described by Vrolijk [Vrolijk, 2023].



#### 4.2.1. THE DATA DIODE TEST SETUP

The Vrolijk setup -as shown in Figure 4.1- consist of two MC210CS TP-Link Gigabit Single-Mode Media Converters, one Single Mode Fiber Optic Splitter, and two Cat6e Ethernet cables.



Figure 4.1: Data diode test setup, implements Figure 3.2.

The data diode hardware is connected to a GMKtec NucBox M5 Plus<sup>1</sup> as shown in Figure 4.2. This NUC is based on an AMD Ryzen 7 5825U CPU and is equipped with two Realtek RTL8125 2.5 Gbps NICs with RJ45 ports supported by the r8169 driver, as well as WiFi 6E, 64 GB of RAM, and a 1 TB SSD for storage.

The NUC is running Kubuntu 24.04.2 LTS (kernel version 6.8.0-58-generic), with all the necessary files installed to compile and build Linux kernel modules written in C. In addition, Python 3.12 is installed to facilitate scripting. Finally, the tools Netcat (OpenBSD version), pydiode 0.0.1<sup>2</sup> [Story, 2023a], tcpdump and Wireshark are also installed. Apart from a few Bash scripts, Python 3.12 is used for all scripting tasks in this research.

#### 4.2.2. THE DATA DIODE NETWORK TESTING SETUP

Without additional network settings, data transfer between the two NICs will take a shortcut through the Linux kernel, bypassing the data diode hardware in the process. To enforce proper routing through the hardware, Linux network namespaces [Lowe, 2013] are used to separate the NICs logically. Listing 4.1 shows the Bash script needed to configure the appropriate networking environment.

<sup>1</sup><https://www.gmktec.com/products/amd-ryzen-7-5825u-mini-pc-nucbox-m5-plus>; The NUC used by Story (Intel NUC 11 Pro – NUC11TNHi50L) was no longer available at the start of this research.

<sup>2</sup>Although the pydiode project is hosted on GitHub, version 0.0.1 is only available from PyPI: <https://pypi.org/project/pydiode/0.0.1/>



Figure 4.2: The GMKtec NucBox M5 plus.

First, two network namespaces are created: *sender\_ns* and *receiver\_ns*. Next, each network interface is moved to its designated namespace: *enp1s0* to *sender\_ns* and *enp2s0* to *receiver\_ns*. In the third step, each interface is assigned a static IP address: *enp1s0* gets 10.0.1.2/24, and *enp2s0* gets 10.0.1.1/24. Step four brings the NICs up, after which a route to the 10.0.1.1 address is added. Finally, because the Address Resolution Protocol (ARP) is unavailable in this setup, a manual ARP entry is added for the receiving IP address.

Executing script 4.1 results in the correct initialization of the NICs for the data diode configuration, leaving only the WiFi connection in the default Linux network namespace.

```
#Sender IF: enp1s0; Receiver - IF: enp2s0 (<MAC address>)

# Creating network namespaces
sudo ip netns add sender_ns
sudo ip netns add receiver_ns

# Moving NICs to namespaces
sudo ip link set enp1s0 netns sender_ns
sudo ip link set enp2s0 netns receiver_ns

# Setting IP addresses
sudo ip netns exec sender_ns ip addr add 10.0.1.2/24 dev enp1s0
sudo ip netns exec receiver_ns ip addr add 10.0.1.1/24 dev enp2s0

# Bring interfaces up
sudo ip netns exec sender_ns ip link set enp1s0 up
sudo ip netns exec receiver_ns ip link set enp2s0 up

# Adding route in sender_ns to enp2s0
sudo ip netns exec sender_ns ip route add 10.0.1.1 dev enp1s0

# Set ARP entry for enp2s0
sudo ip netns exec sender_ns ip neigh add 10.0.1.1 lladdr <MAC address> dev enp1s0
```

Listing 4.1: setup\_dd.sh: Bash script used to setup the data diode networking environment.

### 4.2.3. PERFORMING TESTS

Each configuration of the NUC -i.e. a specific UDP kernel module combined with particular system, network, and data transfer settings- needs to be tested. For each configuration,

1000 trials are conducted at each of the transfer speeds of 1 Gbps, 750,000,000 bps, and 500,000,000 bps. The baseline test was extended with transmission rates of 250,000,000 bps and 100,000,000 bps to allow direct comparison with Story’s published results [Story, 2023b]. With each trial, a 1 Gbit (125 MB) data file consisting solely of ‘a’ characters will be sent through the data diode. This format of the data file is chosen to facilitate easy compression of tcpdump captured files, if required.

```
create test_data

for trial in range(n_trials):
    start_receiver_process
    start_time = current_timestamp
    start_sender_process
    # wait for receiver_process to complete
    end_time = current_timestamp
    transfer_time = end_time - start_time
    compare_checksums_of_sent_and_received_data

calculate CFR over the n_trials performed trials
calculate average transfer_time
```

Listing 4.2: The main test process as pseudo code.

Since testing is conducted by sending a 1 Gbit file, file loss is used as an indicator of packet loss. The SHA256-hashes of the sent and received files will be compared to determine whether any data was lost during transmission. For each series of 1000 trials, the Completed File Ratio (CFR) is calculated using Equation 2.2. With the different CFR values it is possible to compare the effectiveness of each configuration.

To minimize background processing on the NUC running Kubuntu, the WiFi connection is disabled during testing. Additionally, services such as Baloo (the KDE file indexer) and Bluetooth are also disabled.

## NETCAT AND PYDIODE

Data transfer tests are performed using Netcat and Pydiode. In the case of Pydiode, redundancy settings of ‘1’ (no redundancy) and ‘2’ (all data sent twice) are applied.

Pydiode includes the *–max-bitrate* option for controlling the transmission rate, whereas Netcat does not provide such functionality. To address this limitation, Story [Story, 2023b] introduces the Python script *regulator.py*, which enables reliable bitrate limiting when using Netcat. In this study, *regulator.py* is likewise employed to ensure controlled transmission rates.

To prevent the receiving Netcat process from terminating before the sending process has started, the *-w 1* option is used. This option ensures that the receiver terminates if no data is received for one second. Since UDP is connectionless and does not provide an explicit EOF signal, the sender simply stops transmitting datagrams. The receiver has no inherent way of knowing when the transfer is complete; therefore, the *-w* option is required to ensure proper termination of the receiving process. As a result, the measured data transfer time will always be at least one second, and effectively one second longer than the actual transfer duration.

#### 4.2.4. SCRIPTING

The test scripts are built using Python 3.12, implement the basic test process shown in Listing 4.2, and are available from GitHub<sup>3</sup>. These scripts were developed in parallel with the research and also implement the requirements listed below.

1. **Configurability test runs**

All relevant options (i.e. transfer speeds and the data transfer tool to run) can be modified by tweaking the *config.py* module.

2. **Configurability alternative UDP.c kernel module**

Options specific to the alternative UDP.c module can be configured directly from the test scripts. There are no configurable options available for the default UDP.c module.

3. **Transparency data transfer tools**

The scripts handle all responsibilities regarding running the data transfer tools and their options.

4. **Communication with alternative UDP.c kernel module**

The scripts can read and reset certain counters within the alternative UDP.c kernel module.

5. **Network capturing capability**

The test scripts are able to capture network traffic with tcpdump.

6. **Read values from measuring points**

For each trial, these values are read, processed and stored.

7. **Show most important statistics after a series of trials**

After each series of trials, the CFR and the average file transfer duration is shown.

8. **Output data in CSV format**

At the end of each series of trials, a dedicated CSV file for that series is created, containing all data pertaining each trial.

#### 4.2.5. TEST OUTPUT AND ANALYSIS

The test output is stored in CSV files<sup>4</sup>, the structure of which is described in Appendix A. For each tested configuration, a CSV file is created for every transfer speed. The Jupyter Notebooks<sup>5</sup> used for analyzing the CSV files are described in Appendix B. In addition to the CSV files, a testresults.txt file is added for each tested configuration. An example<sup>6</sup> of which is shown below.

---

<sup>3</sup><https://github.com/Gert-Jan1966/DataDiodesAndPacketLoss/tree/main/Testscripts>

<sup>4</sup><https://github.com/Gert-Jan1966/DataDiodesAndPacketLoss/tree/main/Testresults>

<sup>5</sup><https://github.com/Gert-Jan1966/DataDiodesAndPacketLoss/tree/main/Notebooks>

<sup>6</sup><https://github.com/Gert-Jan1966/DataDiodesAndPacketLoss/blob/main/Testresults/Baseline/default/nc/testresults.txt>

```

Current kernel module config:
Default UDP.c kernel module

Test results
-----
Bitrate: 1000000000 bits/second
Success rate: 97.6%
Average duration: 2.07 seconds

Bitrate: 750000000 bits/second
Success rate: 93.3%
Average duration: 2.37 seconds

Bitrate: 500000000 bits/second
Success rate: 97.3%
Average duration: 3.04 seconds

Bitrate: 250000000 bits/second
Success rate: 94.6%
Average duration: 5.05 seconds

Bitrate: 100000000 bits/second
Success rate: 95.0%
Average duration: 11.03 seconds

```

Listing 4.3: An example of a testresults.txt file, containing high level statistics.

#### 4.2.6. RECAPITULATION OF THE TEST SETUP

The implemented test setup is a realization of the diagram shown in Figure 3.2 in Section 3.2.1. Within the test setup for this research, the NUC functions as both the Sending and Receiving Proxy.

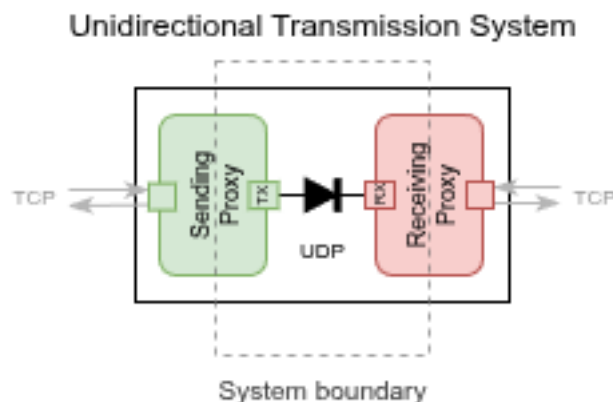


Figure 4.3: System boundary of the test setup.

Compared to the architecture shown in Figure 2.5, the system boundary of the test setup is illustrated in Figure 4.3. The focus is therefore on the transmission of data by the Sending Proxy using UDP over the unidirectional connection and the reception of this data by the Receiving Proxy.

# 5

## UDP MODULES

This chapter outlines the functionality of the default Linux UDP.c kernel module and examines how it can be tuned for use in data diode environments.

### 5.1. THE UDP.C KERNEL MODULE

The default UDP.c Linux kernel module is a critical component of the Linux networking stack, as it implements one of the core protocols used in both local networks and on the Internet. UDP.c offers the key functionality listed below [Damato, 2016, 2017]:

1. **Socket Management**

The module enables applications to create and manage UDP sockets. It allows applications to bind to specific ports, facilitating communication with remote hosts via designated port numbers.

2. **Socket Buffer Management**

The UDP socket buffers are limited in size and directly influence performance. Queue management within the module ensures fair access among multiple sockets and helps maintain responsiveness under load.

3. **Datagram Transmission**

For outgoing data, the module constructs the UDP header. The completed datagram is then passed to the Internet Layer for transmission.

4. **Datagram Reception**

For incoming data, the module examines the destination port of each UDP datagram to route it to the correct socket. If a checksum is present, it is verified to ensure the datagram's integrity. If the checksum fails or no socket is bound to the destination port, the datagram is dropped.

5. **Multicasting and Broadcasting**

The module supports sending datagrams to multiple recipients simultaneously, enabling efficient distribution of data to groups or entire networks.

6. **Error Handling**

The module manages errors such as invalid checksums or unbound ports by discarding erroneous datagrams and, in some cases, sending ICMP error messages back to the sender.



The UDP.c module implements UDP's connectionless nature, offering no mechanisms for connection establishment, flow control, or guaranteed delivery. This simplicity makes UDP fast and efficient. The responsibility for detecting and recovering from transmission errors lies with higher protocol layers or the application itself.

### 5.1.1. MODIFYING THE UDP.C MODULE

The UDP.c module in the Linux kernel is not easily replaceable due to its critical role and deep integration within the networking stack. It is tightly coupled with the Internet Layer, socket APIs, and network drivers [Damato, 2017].

Any replacement must remain compatible with existing interfaces, replicate performance optimizations such as checksum offloading to NICs (when UDP checksumming is enabled), and preserve essential security mechanisms, such as error handling. Furthermore, the complexity of the codebase and the need for extensive validation make replacing UDP.c a highly challenging task.

Because UDP is a core (inter)networking protocol, and given the monolithic nature of the Linux kernel [Wikipedia, 2025a], the UDP.c module is compiled directly into the kernel itself. This is evidenced by the Makefile found in the *net/ipv4/* directory of the Linux kernel source code project [Torvalds, 2025]. Therefore it is not possible to replace this kernel module with another implementation on a *running* Linux system. Nevertheless, a new and reloadable kernel module has been developed. This is described in the following section.

## 5.2. THE DD\_UDP.C MODULE

There are other ways in which the behavior of a kernel module can be influenced. A new additional *dd\_udp.c* kernel module has been created<sup>1</sup>. This alternative module is not meant as a direct replacement for the standard UDP.c module. It is rather a custom overlay designed for a specific purpose: implementing a UDP-based data diode with enhanced features like packet loss and corruption tracking for use with testing, as well as awareness of hardware checksum offloading.

### 5.2.1. FEATURES AND CAPABILITIES OF DD\_UDP.C

#### DATA DIODE FUNCTIONALITY

A data diode must enforce strict unidirectional data flow from sender to receiver. This prevents any reverse communication that could compromise security. The *dd\_udp.c* module leverages *Netfilter hooks* to accomplish this goal.

Netfilter hooks are predefined points in the Linux kernel's networking stack where kernel modules (such as firewalls or custom code) can intercept, modify, or drop network packets [Thummaluru, 2024]. These hooks allow developers to insert custom logic at various stages of packet processing, such as when a packet enters or leaves the system.

The *sender\_hook* function filters outgoing traffic on the sender interface, accepting only UDP datagrams and dropping everything else, like TCP or ICMP traffic. The *receiver\_hook* processes incoming UDP datagrams on the receiver interface.

---

<sup>1</sup>[https://github.com/Gert-Jan1966/DataDiodesAndPacketLoss/tree/main/dd\\_udp-module](https://github.com/Gert-Jan1966/DataDiodesAndPacketLoss/tree/main/dd_udp-module)

## A LOADABLE KERNEL MODULE (LKM)

The `dd_udp.c` module is implemented as a Loadable Kernel Module (LKM) [Wikipedia, 2025b], allowing it to be inserted using the `insmod` and removed with `rmmod` Linux commands. To create an LKM, the developer writes a C source file that includes the necessary kernel headers (e.g., `<linux/module.h>`, `<linux/init.h>`) and define `init` and `exit` functions using `module_init()` and `module_exit()`. After compiling the source file, the resulting `.ko` file can be loaded with `insmod`.

## NETWORK NAMESPACE SUPPORT

Network namespace support [Lowe, 2013] provides isolation of the sender and receiver in separate network namespaces preventing unintended cross-talk between interfaces. This allows the module to operate in a controlled, isolated environment.

The module retrieves namespaces via file descriptors (`sender_ns_fd`, `receiver_ns_fd`) during initialization. Netfilter hooks are registered within these namespaces, ensuring traffic filtering applies only to the specified interfaces.

```
$ sudo ./load_diode -n sender_ns receiver_ns enp1s0 enp2s0

Module loaded with sender_ns=sender_ns (fd=3), receiver_ns=receiver_ns (fd=4),
sender_ifname=enp1s0, receiver_ifname=enp2s0, verbose_netlink=0
```

Listing 5.1: Loading the `dd_udp.c` kernel module using the `load_diode` utility.

## DATAGRAM COUNTING

Monitoring the number of sent, received, and corrupted UDP datagrams is essential for verifying the diode's operation. Detecting packet loss and ensuring data transfer reliability, especially in a one-way system where sending acknowledgements is not an option.

64 Bit atomic counters (`sent_datagrams`, `received_datagrams` and `corrupted_datagrams`) are incremented in the `sender_hook` and `receiver_hook` functions. These counters are thread-safe, ensuring accurate tracking in the kernel's concurrent environment.

```
[ 80.351766] dd_udp: loading out-of-tree module taints kernel.
[ 80.351772] dd_udp: module verification failed: signature and/or required key
missing - tainting kernel
[ 80.352160] dd_udp: Registered sender hook for sender FD 3, sender_if enp1s0
[ 80.352177] dd_udp: Registered receiver hook for receiver FD 4, receiver_if enp2s0
[ 80.352179] dd_udp: Loaded with sender FD 3 -> receiver FD 4, sender_if enp1s0,
receiver_if enp2s0
```

Listing 5.2: Dmesg output after loading the `dd_udp.c` kernel module.

## CHECKSUM VERIFICATION

Since UDP does not inherently guarantee data integrity, enabling checksum verification can help a data diode to detect packet corruption in scenarios where no feedback mechanism exists to request retransmission.

The `receiver_hook` verifies UDP checksums if enabled (`enable_checksum = 1`). It supports both computation by the module (using the `csum_tcpudp_nofold()` and `csum_partial()` functions from the `checksum.h` header file) and NIC offloading (checking `skb->ip_summed`). Corrupted datagrams increment the `corrupted_datagrams` counter, providing insight into data integrity issues.



## NETLINK COMMUNICATION

Netlink communication allows for user-space programs to interact with the kernel module [docs.kernel.org, 2025]. It also enables real-time monitoring and configuration without requiring module reloads or kernel recompilation, enhancing usability and flexibility.

The module uses a Netlink socket with commands such as `NL_CMD_RESET_COUNTERS`, `NL_CMD_QUERY_COUNTERS`, and `NL_CMD_SET_CHECKSUM`. The `nl_input` function processes these commands, while `nl_send` sends responses or logs to user-space. These commands are implemented for measurement, troubleshooting, and testing purposes.

### 5.2.2. FUNCTIONALITY OMITTED IN DD\_UDP.C

This section describes the functionality omitted in `dd_udp.c`.

#### UNNECESSARY FUNCTIONALITY

##### ICMP support

ICMP (Internet Control Message Protocol) is primarily used for diagnostics and error reporting, like "destination unreachable". In a data diode setup, no reverse traffic is possible by design. Because the TCP proxies on either side of the diode cannot exchange messages bidirectionally, ICMP support between the proxies is unnecessary.

##### Addressing support

In the data diode setup, the communication path is fixed and unidirectional, connecting a known sender directly to a known receiver. Since there is only one valid destination, the sending proxy does not need to perform dynamic addressing or resolve multiple targets. This eliminates the need for address discovery, routing, or destination management logic. Statically configured addressing is sufficient and more appropriate for such a constrained environment.

#### FUNCTIONALITY INVOKED BY THE DEFAULT UDP.C MODULE

##### Socket management

The `dd_udp.c` kernel module relies on the standard `UDP.c` module for socket management. `dd_udp.c` is designed to intercept and process UDP datagrams at the Internet Layer using Netfilter hooks, not to handle socket-level operations like creation, binding, or data transmission. By leveraging `UDP.c`, `dd_udp.c` avoids duplicating existing functionality and focuses solely on datagram-level tasks like filtering and checksum handling.

##### Buffer handling

The `dd_udp.c` module also relies on `UDP.c` for buffer handling because it intercepts UDP datagrams using Netfilter hooks, allowing it to process existing socket buffers (`sk_buffs`) without needing to manage their allocation or deallocation. For outgoing datagrams, `UDP.c` allocates and prepares the `sk_buff`, which `dd_udp.c` can then inspect or modify at the hook point. Similarly, for incoming datagrams, `dd_udp.c` acts on the `sk_buff` before `UDP.c` handles its delivery, leveraging the kernel's buffer management to ensure efficiency and compatibility.

### 5.2.3. EXPECTATIONS FOR THE DD\_UDP.C MODULE

The `dd_udp.c` module is a custom Linux kernel module designed to enforce unidirectional UDP traffic between network namespaces. It functions as an overlay to the default `UDP.c`

module. `dd_udp.c` leverages Netfilter hooks to intercept and manage UDP datagrams on designated interfaces. The module tracks key statistics—the number of sent, received, and corrupted datagrams—while supporting checksum verification with consideration for hardware offloading. It also provides a Netlink interface for user-space communication, enabling runtime configuration and monitoring.

`dd_udp.c` introduces some overhead due to additional processing in the Netfilter hooks (e.g., checksum computation, counter updates, and Netlink logging). However, it omits ICMP handling and dynamic addressing entirely. It is expected that, when UDP checksumming and Netlink logging are disabled, the performance of `dd_udp.c` will match—or slightly exceed—that of the default `UDP.c` module.

Netcat	default UDP.c		dd_udp.c	
Bitrate	CFR	Avg. Duration	CFR	Avg. Duration
100 Mbps	095.0 %	11.03 s	095.6 %	11.04 s
250 Mbps	094.6 %	05.05 s	093.9 %	05.06 s
500 Mbps	097.3 %	03.04 s	096.8 %	03.04 s
750 Mbps	093.3 %	02.37 s	094.3 %	02.39 s
1 Gbps	097.6 %	02.07 s	095.6 %	02.08 s

Table 5.1: Initial Netcat test results with default `UDP.c` and `dd_udp.c` kernel modules.

pydiode		default UDP.c		dd_udp.c	
Bitrate	Redundancy	CFR	Avg. Duration	CFR	Avg. Duration
100 Mbps	1	100.0 %	10.43 s	100.0 %	10.35 s
250 Mbps	1	100.0 %	04.28 s	100.0 %	04.24 s
500 Mbps	1	099.9 %	02.21 s	100.0 %	02.19 s
750 Mbps	1	091.3 %	01.56 s	089.3 %	01.53 s
1 Gbps	1	060.1 %	01.35 s	062.5 %	01.33 s
100 Mbps	2	100.0 %	20.44 s	100.0 %	20.44 s
250 Mbps	2	100.0 %	08.31 s	100.0 %	08.31 s
500 Mbps	2	100.0 %	04.35 s	100.0 %	04.32 s
750 Mbps	2	100.0 %	03.01 s	100.0 %	02.96 s
1 Gbps	2	100.0 %	<b>02.18 s</b>	100.0 %	<b>02.18 s</b>

Table 5.2: Initial pydiode test results with default `UDP.c` and `dd_udp.c` kernel modules.

### 5.3. BASELINE TEST RESULTS FOR BOTH KERNEL MODULES

To establish a baseline, tests were conducted using both kernel modules. For the `dd_udp.c` module, UDP checksumming and Netlink logging were disabled to allow a fair comparison with the default `UDP.c` module. Table 5.1 presents the results using Netcat, while Table 5.2 shows the results using pydiode, tested with both a redundancy level of 1 and 2.

The results of the Netcat tests show almost no difference in average transmission times, and a CFR of 100 % is never achieved. The tests with pydiode also show comparable average transmission times for both modules, with `dd_udp.c` sometimes performing slightly

faster. The fastest average transmission time with a CFR of 100 % is achieved using pydiode with a redundancy of 2, where both modules perform equally well.

# 6

## MEASURING POINTS

While conducting research and working on the alternative *dd\_udp.c* kernel module, the measure points as listed below were found. However, not all of these measure points proved to be useful within the context of this research. This will be discussed later on in this chapter.

### 6.1. RELEVANT MEASURING POINTS

#### 6.1.1. FILE INTEGRITY - APPLICATION LAYER (LAYER 4)

As discussed in Section 4.2.3, SHA-256 hashes of the transmitted and received files are used to detect violations of file integrity. Since these checks are performed at the Application Layer and are not part of the network stack, they will not be discussed further in this chapter.

#### 6.1.2. SOCKETS - TRANSPORT LAYER (LAYER 3)

Within the Transport Layer, UDP datagrams can be counted by accessing */proc/net/snmp* on the Linux filesystem. The */proc/net/snmp* file provides a real-time snapshot of various network protocol statistics as defined by the SNMP (Simple Network Management Protocol) MIB-II standard [IETF, 1991]. It includes counters for protocols such as IP, ICMP, TCP, and UDP, reflecting metrics like packet counts, errors, and dropped connections.

```
Ip: Forwarding DefaultTTL InReceives InHdrErrors InAddrErrors ForwDatagrams ...
Ip: 1 64 1923048 0 12 43005 ...

Icmp: InMsgs InErrors InDestUnreachs InTimeExcds InParmProbs ...
Icmp: 412 0 298 64 0 ...

Tcp: RtoAlgorithm RtoMin RtoMax MaxConn ActiveOpens PassiveOpens AttemptFails ...
Tcp: 1 200 120000 -1 30541 13192 247 ...

Udp: InDatagrams NoPorts InErrors OutDatagrams RcvbufErrors SndbufErrors ...
Udp: 382492 5 0 392304 0 1 ...
```

Listing 6.1: An example layout of the */proc/net/snmp* file.

Listing 6.1 shows that each protocol section begins with a line of field names, followed by a line of corresponding values, enabling userspace tools and administrators to monitor

and analyze the network stack's behavior over time. Linux maintains a dedicated copy of this file for each network namespace.

From this `/proc/net/snmp` file, the following UDP statistics are retrieved: *OutDatagrams*, *SndbufErrors*, *InDatagrams*, *InErrors* and *RcvbufErrors* [Damato, 2017]. These counters are set to zero when the system boots.

### 6.1.3. KERNEL MODULE - TRANSPORT/INTERNET LAYER (LAYERS 3/2)

At this level, the number of UDP datagrams are measured by the `dd_udp.c` kernel module. The provided counters are updated leveraging the Netfilter `NF_INET_LOCAL_OUT` and `NF_INET_LOCAL_IN` hooks. These counters - *sent\_datagrams*, *received\_datagrams* and *corrupted\_datagrams*- are implemented using threadsafe `atomic64_t` variables and can be reset or queried from user programs using Netlink commands.

The *corrupted\_datagrams* counter is incremented whenever a datagram has an invalid UDP checksum (assuming checksum verification is enabled).

### 6.1.4. LINK LAYER (LAYER 1)

On the Link Layer, two distinct measuring points are used.

#### SYSFS FILES

The Linux kernel offers the `/sys/class/net/<iname>/statistics/` directory structure, in which it exports information about the network interfaces [kernel.org, 2024]. This directory contains the *tx\_dropped*, *tx\_errors*, *rx\_dropped* and *rx\_errors* files. Each of these files only hold a numerical value. These statistics are directly tied to the receive and send buffers of the NIC's. They are reset to zero only when the system is rebooted or when the network interface is removed and re-added.

#### TCPDUMP

*Tcpdump* is a command-line network packet analyzer that captures and displays traffic passing through a network interface [Tcpdump.Group, 2025]. It allows users to filter packets based on protocols, IP addresses, ports, and other criteria, making it a powerful tool for network diagnostics and security analysis. *tcpdump* provides real-time visibility into network behavior and can also save captures in `.pcap` format for later analysis with tools like Wireshark or Python-based frameworks. Within the context of this research: *tcpdump* captures Ethernet frames, when they leave a sending NIC or enter a receiving NIC.

## 6.2. SENDING PROXY

In the following description, the path of the data within the sender is traced through layers of the Internet Model. Along the way, the forementioned measurement points are positioned, as shown Figure 6.1.

### 6.2.1. TRANSPORT LAYER

These measurement points, which measure statistics concerning the UDP sender socket buffer, are located within the `/proc/net/snmp` file [Damato, 2017].

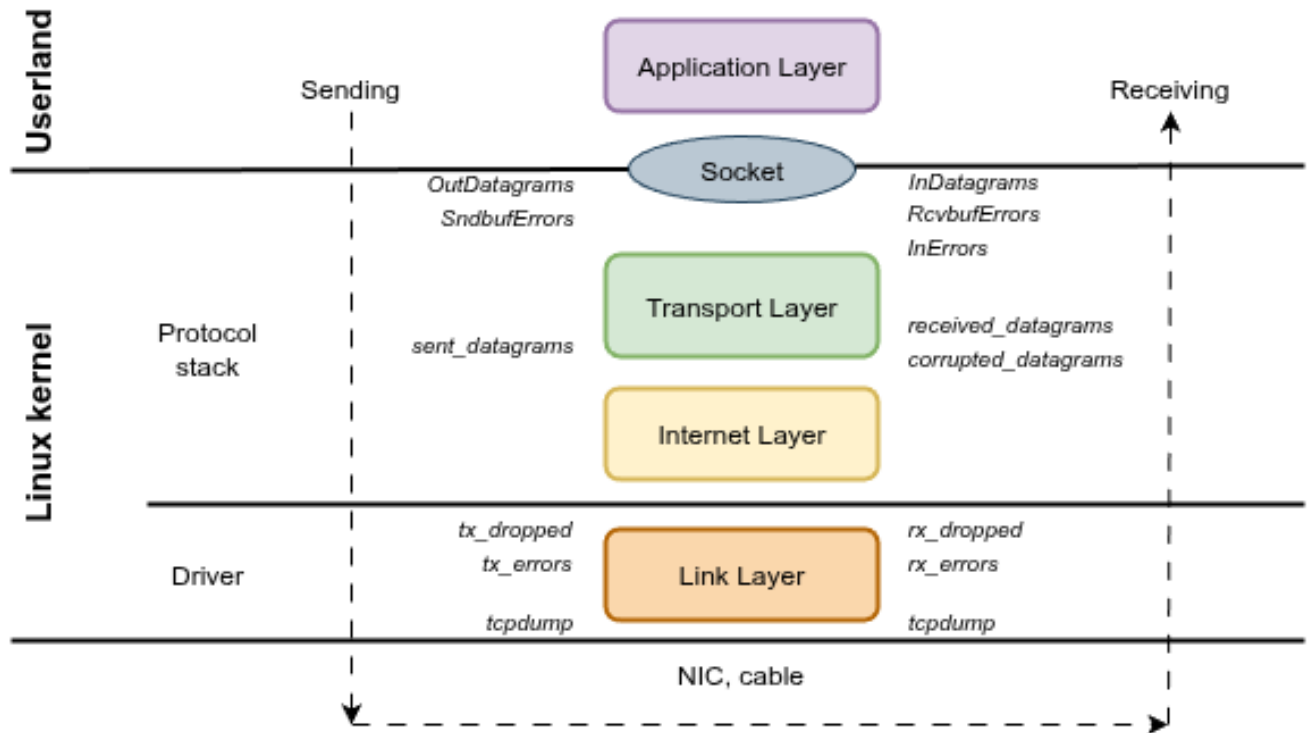


Figure 6.1: Mapping the relevant measuring points on the Internet Model (derived from [Maxnilz, 2025]).

### OUTDATAGRAMS

The *OutDatagrams* value in the `/proc/net/snmp` file (under the UDP section) represents the total number of UDP datagrams that the system has successfully sent, within a particular network namespace. It only counts datagrams handed off for transmission and does not include those that were discarded due to errors. This counter provides insight into the volume of outbound UDP traffic generated by the system.

### SNDBUFERRORS

*SndbufErrors* in `/proc/net/snmp` under the UDP section represents the number of data segments that failed to be sent because the socket's send buffer was full. This may occur when the application sends data faster than the kernel can transmit it over the network. A high number of *SndbufErrors* may indicate insufficient buffer sizing or network congestion affecting transmission performance.

### 6.2.2. DD\_UDP.C MODULE: TRANSPORT (UDP) AND INTERNET (IP) LAYERS

On the sender side, one metric is recorded within the `dd_udp.c` module.

### SENT\_DATAGRAMS

All datagrams intended to be sent pass the `NF_INET_LOCAL_OUT` Netfilter hook. Within this hook, the *sent\_datagrams* counter is incremented for each datagram after processing but before it is handed to the Link Layer. This reflects the number of datagrams the kernel intends to send.

### 6.2.3. LINK LAYER

At the Link Layer of the sender proxy, three measuring points can be utilized.

#### SYSFS: TX\_ERRORS

Tx\_errors in sysfs represents the total number of frames that the system attempted to transmit but failed due to errors. These errors can be caused by issues such as carrier problems, collisions, or hardware faults reported by the network device driver. This counter helps assess the reliability and health of the transmission path on a given interface. In short: tx\_errors counts frames that failed to transmit due to hardware or Link Layer errors.

#### SYSFS: TX\_DROPPED

Tx\_dropped indicates the number of outbound frames that were dropped before being transmitted by the network interface. These drops usually occur due to resource limitations, such as insufficient buffer space in the kernel or the network driver. tx\_dropped counts frames that were dropped before reaching the hardware. A high tx\_dropped value may suggest congestion or configuration issues affecting outbound traffic.

#### TCPDUMP - PCAP FILES

Tcpdump captures frames as they leave the NIC, showing the raw frames as they are transmitted over the wire.

## 6.3. RECEIVING PROXY

This section describes the path of the data within the receiver, traced through layers of the Internet Model, as shown Figure 6.1.

### 6.3.1. LINK LAYER

At the Link Layer of the receiver proxy, three measuring points can be utilized.

#### TCPDUMP - PCAP FILES

Tcpdump captures frames as they enter the NIC, showing the raw frames as they are received from the wire, before any kernel processing.

#### SYSFS: RX\_ERRORS

Rx\_errors in sysfs represents the total number of frames received by the network interface that were discarded due to errors. These errors may include CRC errors, frame alignment issues, or other hardware-level problems that prevent proper packet reception. This counter helps diagnose link quality and hardware reliability on the receiving side.

#### SYSFS: RX\_DROPPED

Rx\_dropped indicates the number of incoming frames that were dropped by the network interface before being handed off to the network stack. These drops typically occur due to insufficient buffer space, high system load, or driver limitations. Unlike rx\_errors, which reflect hardware-level faults, rx\_dropped points to resource constraints or software-layer bottlenecks in frame processing.

### 6.3.2. DD\_UDP.C MODULE: TRANSPORT (UDP) AND INTERNET (IP) LAYERS

On the receiver side, two metrics are recorded within the `dd_udp.c` module.

#### RECEIVED\_DATAGRAMS

All received datagrams pass the `NF_INET_LOCAL_IN` Netfilter hook. Leveraging this hook, `received_datagrams` counts the datagrams after reassembly and basic validation, before they are passed to the Transport Layer.

#### CORRUPTED\_DATAGRAMS

When UDP checksumming is enabled, the `corrupted_datagrams` counter is incremented for each datagram failing its checksum at the `NF_INET_LOCAL_IN` hook.

### 6.3.3. TRANSPORT LAYER (UDP)

These measurement points collect statistics concerning the UDP receiver socket buffer. These are available from the `/proc/net/snmp` file [Damato, 2017].

#### IN DATAGRAMS

`InDatagrams` in the `/proc/net/snmp` file under the UDP section represents the total number of UDP datagrams delivered to UDP sockets on the system, within a particular network namespace. This includes all packets received successfully, excluding those dropped due to errors such as checksum failures or unavailable ports. It serves as an indicator of incoming UDP traffic that was processed by the kernel and passed to user-space applications.

#### IN ERRORS

The `InErrors` value in `/proc/net/snmp` under the UDP section indicates the number of incoming UDP datagrams that were received but could not be delivered to a socket due to errors. Common causes include checksum failures, malformed packets, or resource limitations. This counter helps identify issues in the reliability of incoming UDP traffic. This metric includes the errors from the next metric.

#### RCVBUF ERRORS

`RcvbufErrors` located in the UDP section of `/proc/net/snmp` represents the number of incoming UDP packets that were dropped because the socket receive buffer was full. This typically occurs when data arrives faster than the receiving application can process it, leading to buffer exhaustion. A high `RcvbufErrors` count may indicate insufficient socket buffer size or performance bottlenecks in the receiving application.

## 6.4. MEASURING POINTS AND TEST SCRIPTING

This section describes how the test scripts make use of the measuring points.

#### UDP SOCKETS - /PROC/NET/SNMP

The statistics of the UDP sockets are retrieved and calculated by the `dd_transport_layer_stats.py`<sup>1</sup> script. The five retrieved metrics (`OutDatagrams`, `SndbufErrors`, `InDatagrams`, `InErrors`, and `RcvbufErrors`) must be read both at the start and at the end of a trial in order to correctly determine the changes that occurred during the execution of that trial.

<sup>1</sup>[[https://github.com/Gert-Jan1966/DataDiodesAndPacketLoss/blob/main/Testscripts/dd\\_transport\\_layer\\_stats.py](https://github.com/Gert-Jan1966/DataDiodesAndPacketLoss/blob/main/Testscripts/dd_transport_layer_stats.py)]



## DATAGRAM COUNTERS IN THE DD\_UDP.C MODULE

The `dd_udp.c` module is able to track three datagram counters: `sent_datagrams`, `received_datagrams`, and `corrupted_datagrams`. These counters can be reset and read using the `dd_netlink.py`<sup>2</sup> script. When UDP checksumming is disabled via `dd_config.py`<sup>3</sup>, `corrupted_datagrams` will return “na” (not available). The default UDP.c module does not provide these counters; when testing with UDP.c, the script always returns “na” for each of these three counters during test runs.

## LINK LAYER STATISTICS WITH SYSFS

The four sysfs counters (`rx_errors`, `rx_dropped`, `tx_errors`, and `tx_dropped`) are tracked by the `dd_link_layer_stats.py`<sup>4</sup> script. These counters must be read both at the start and at the end of each trial in order to correctly determine the changes that occurred during the execution of that trial.

## PACKET CAPTURING - TCPDUMP

If packet capturing is enabled, tcpdump is started and stopped by the `dd_capture.py`<sup>5</sup> script. Its activation and options can be configured in the `dd_config.py` script.

## 6.5. OBSERVATIONS

The tests conducted produced the following results.

1. In the `dd_udp.c` module, `sent_datagrams` consistently equals `received_datagrams`.
2. There is no packet loss within the data diode hardware.
3. Data loss occurs only at the UDP receiver socket buffer.
4. The `/proc/net/snmp` values for `InErrors` and `RcvbufErrors` are always identical.
5. When using `pydiode`, `SndbufErrors` > 0 (in `/proc/net/snmp`) is observed at transfer speeds of 750 Mbps and 1 Gbps, even when all files are received correctly.

Not all identified measurement points turned out to be useful; these points are discussed in the following sections.

### 6.5.1. THE USEFULNESS OF UDP CHECKSUMMING

During the preparatory tests conducted for this research, no received UDP datagrams were ever rejected on the basis of their UDP checksum. As noted in Chapter 2, erroneous Ethernet frames will probably fail (and get dropped) due to their CRC-32 values before the UDP checksum calculation is even performed.

Although UDP checksumming showed no impact on throughput, the published tests were performed with UDP checksumming disabled, in order to minimize its influence on the measured throughput results.

### 6.5.2. INERRORS VERSUS RCVBUFERRORS

During the tests, the values of `InErrors` and `RcvbufErrors` are always the same. `RcvbufErrors` increments when a datagram is dropped because the socket’s receive buffer is full. `InErrors`

<sup>2</sup>[https://github.com/Gert-Jan1966/DataDiodesAndPacketLoss/blob/main/Testscripts/dd\\_netlink.py](https://github.com/Gert-Jan1966/DataDiodesAndPacketLoss/blob/main/Testscripts/dd_netlink.py)

<sup>3</sup>[https://github.com/Gert-Jan1966/DataDiodesAndPacketLoss/blob/main/Testscripts/dd\\_config.py](https://github.com/Gert-Jan1966/DataDiodesAndPacketLoss/blob/main/Testscripts/dd_config.py)

<sup>4</sup>[https://github.com/Gert-Jan1966/DataDiodesAndPacketLoss/blob/main/Testscripts/dd\\_link\\_layer\\_stats.py](https://github.com/Gert-Jan1966/DataDiodesAndPacketLoss/blob/main/Testscripts/dd_link_layer_stats.py)

<sup>5</sup>[https://github.com/Gert-Jan1966/DataDiodesAndPacketLoss/blob/main/Testscripts/dd\\_capture.py](https://github.com/Gert-Jan1966/DataDiodesAndPacketLoss/blob/main/Testscripts/dd_capture.py)

also counts datagrams discarded due to buffer overflows, so when buffer exhaustion is the only source of receive errors, both counters increase in sync, resulting in InErrors equalling RcvbufErrors.

### **6.5.3. IMPACT OF TCPDUMP**

During the preparatory tests, it was also found that capturing Ethernet frames worked correctly, but placed an excessive load on the resources of the NUC, primarily degrading CPU capacity. While the average data throughput was not affected, the CFR clearly decreased. The published tests were therefore conducted without the use of tcpdump.

### **6.5.4. SNDBUFERRORS > 0 WITH PYDIODE DESPITE CORRECT FILE RECEPTION**

Only when using pydiode, the measurements show that SndbufErrors increase during file transfers. This also occurs when files are received correctly. The effect is most noticeable at a transmission speed of 1 Gbps, to a much lesser extent at 750 Mbps, and is hardly observed at 500 Mbps.

The phenomenon is particularly interesting for pydiode with a redundancy setting of 1 at a transfer speed of 1 Gbps. Since each data segment is transmitted only once, data loss should directly result in corrupted files. However, even with nonzero SndbufErrors, the files are still received intact. Appendix D presents the observed increase in SndbufErrors across different test scenarios using pydiode with the redundancy option set to 1.

A possible explanation for this phenomenon is the use of the asyncio library by pydiode. By sending data more efficiently, the load on the UDP send socket buffer increases, potentially causing overflow and a rise in SndbufErrors. However, this does not explain why the files arrive intact at the receiver. An initial review of both pydiode and asyncio provided no indication that data is retransmitted in the event of errors or exceptions. The asyncio functionality used appears solely to send the provided data as UDP datagrams.

Both the sending and receiving NICs operate within their dedicated Linux network namespaces. The Python test scripts invoke the sending and receiving pydiode processes with awareness of these namespaces. This ensures that no other processes within the Kubuntu environment can access the UDP sender socket.

Although the files are received correctly and the other measurements recorded in the CSV result files appear normal, further investigation is required, as the issue occurs at the first usable measurement point when sending UDP data through the data diode.

# 7

## MODIFYING INTERNAL PARAMETERS OF THE DATA DIODE PROXIES

This chapter describes the impact on packet loss, resulting from modifying several parameters within the data diode proxies. Specifically, it examines increasing the UDP receive socket buffer size, increasing the MTU value, and adjusting the size of the transmitted data segments to match the modified MTU.

### 7.1. THE UDP RECEIVING PROXY SOCKET BUFFER

Section 6.5 mentions the fact that no data is lost while traversing the data diode hardware. Data loss occurs only at the UDP socket buffer of the receiving proxy. This buffer is part of the socket used by the receiving process to temporarily store incoming data until it is read by the application. Increasing this buffer size allocates more memory for holding incoming data segments, thereby reducing the likelihood of packet loss when data arrives faster than the application can process it.

```
# Setting the max and default sizes used by UDP receiver socket buffers to 4 MiB.
sudo sysctl -w net.core.rmem_max=4194304
sudo sysctl -w net.core.rmem_default=4194304
```

Listing 7.1: Bash script used increase the UDP receiver socket buffer size to 4 MiB.

The operating system enforces limits on socket buffer sizes. On Linux, the default maximum receive buffer size is typically 208 KiB. These limits are governed by kernel parameters that can be adjusted using the `sysctl` interface, as shown in Listing 7.1. For this research, tests are conducted for receiver buffer sizes of 1, 2, 4, 8 and 16 MiB.

Recv Socket Buffer	Netcat	default UDPc		dd_udp.c	
Size	Bitrate	CFR	Avg. Duration	CFR	Avg. Duration
Default	500 Mbps	097.3 %	03.04 s	096.8 %	03.04 s
Default	750 Mbps	093.3 %	02.37 s	094.3 %	02.39 s
Default	1 Gbps	097.6 %	02.07 s	095.6 %	02.08 s
1 MiB	500 Mbps	099.4 %	03.04 s	098.4 %	03.03 s
1 MiB	750 Mbps	095.4 %	02.38 s	095.5 %	02.39 s
1 MiB	1 Gbps	099.5 %	02.07 s	098.6 %	02.08 s
2 MiB	500 Mbps	099.3 %	03.04 s	099.4 %	03.06 s
2 MiB	750 Mbps	095.6 %	02.38 s	095.8 %	02.37 s
2 MiB	1 Gbps	099.6 %	02.07 s	098.9 %	02.09 s
4 MiB	500 Mbps	098.7 %	03.08 s	098.9 %	03.03 s
4 MiB	750 Mbps	095.1 %	02.40 s	096.3 %	02.37 s
4 MiB	1 Gbps	097.6 %	02.10 s	099.5 %	02.08 s
8 MiB	500 Mbps	098.8 %	03.06 s	099.3 %	03.04 s
8 MiB	750 Mbps	096.6 %	02.38 s	096.8 %	02.37 s
8 MiB	1 Gbps	099.6 %	02.07 s	099.5 %	02.08 s
16 MiB	500 Mbps	098.7 %	03.04 s	098.7 %	03.06 s
16 MiB	750 Mbps	097.2 %	02.38 s	098.5 %	02.39 s
16 MiB	1 Gbps	098.1 %	02.08 s	099.2 %	02.09 s

Table 7.1: Adjusted receiver socket buffers: Netcat test results.

Recv Socket Buffer	pydiode	default UDPc		dd_udp.c	
Size	Bitrate	CFR	Avg. Duration	CFR	Avg. Duration
Default	500 Mbps	099.9 %	02.21 s	100.0 %	02.19 s
Default	750 Mbps	091.3 %	01.56 s	089.3 %	01.53 s
Default	1 Gbps	060.1 %	01.35 s	062.5 %	01.33 s
1 MiB	500 Mbps	100.0 %	02.23 s	100.0 %	02.19 s
1 MiB	750 Mbps	100.0 %	01.53 s	100.0 %	01.56 s
1 MiB	1 Gbps	099.6 %	01.40 s	099.8 %	01.38 s
2 MiB	500 Mbps	100.0 %	02.26 s	100.0 %	02.22 s
2 MiB	750 Mbps	100.0 %	01.65 s	100.0 %	01.54 s
2 MiB	1 Gbps	100.0 %	01.46 s	100.0 %	01.38 s
4 MiB	500 Mbps	100.0 %	02.21 s	100.0 %	02.22 s
4 MiB	750 Mbps	100.0 %	01.57 s	100.0 %	01.57 s
4 MiB	1 Gbps	100.0 %	01.38 s	100.0 %	01.40 s
8 MiB	500 Mbps	100.0 %	02.22 s	100.0 %	02.20 s
8 MiB	750 Mbps	100.0 %	01.54 s	100.0 %	01.55 s
8 MiB	1 Gbps	100.0 %	<b>01.34 s</b>	100.0 %	01.36 s
16 MiB	500 Mbps	100.0 %	02.27 s	100.0 %	02.22 s
16 MiB	750 Mbps	100.0 %	01.64 s	100.0 %	01.55 s
16 MiB	1 Gbps	100.0 %	01.38 s	100.0 %	01.44 s

Table 7.2: Adjusted receiver socket buffers: pydiode test results with redundancy=1.

Recv Socket Buffer	pydiode	default UDP.c		dd_udp.c	
Size	Bitrate	CFR	Avg. Duration	CFR	Avg. Duration
Default	500 Mbps	100.0 %	04.35 s	100.0 %	04.32 s
Default	750 Mbps	100.0 %	03.01 s	100.0 %	02.96 s
Default	1 Gbps	100.0 %	02.18 s	100.0 %	02.18 s
1 MiB	500 Mbps	100.0 %	04.36 s	100.0 %	04.31 s
1 MiB	750 Mbps	100.0 %	03.01 s	100.0 %	02.98 s
1 MiB	1 Gbps	100.0 %	02.18 s	100.0 %	<b>02.17 s</b>
2 MiB	500 Mbps	100.0 %	04.34 s	100.0 %	04.32 s
2 MiB	750 Mbps	100.0 %	03.01 s	100.0 %	02.99 s
2 MiB	1 Gbps	100.0 %	02.21 s	100.0 %	02.20 s
4 MiB	500 Mbps	100.0 %	04.38 s	100.0 %	04.31 s
4 MiB	750 Mbps	100.0 %	03.04 s	100.0 %	03.00 s
4 MiB	1 Gbps	100.0 %	02.26 s	100.0 %	02.21 s
8 MiB	500 Mbps	100.0 %	04.43 s	100.0 %	04.39 s
8 MiB	750 Mbps	100.0 %	03.16 s	100.0 %	03.09 s
8 MiB	1 Gbps	100.0 %	02.30 s	100.0 %	02.29 s
16 MiB	500 Mbps	100.0 %	04.39 s	100.0 %	04.34 s
16 MiB	750 Mbps	100.0 %	03.19 s	100.0 %	03.05 s
16 MiB	1 Gbps	100.0 %	02.27 s	100.0 %	02.31 s

Table 7.3: Adjusted receiver socket buffers: pydiode test results with redundancy=2.

### 7.1.1. OBSERVATIONS

The results of the Netcat tests, shown in Table 7.1, reveal almost no differences in average transmission times compared to the baseline results. In all cases, the Completed File Ratio (CFR; see Equation 2.2) benefits from buffer sizes larger than the default size or 208 KiB; however, a CFR of 100 % is never achieved.

The tests with pydiode (with a redundancy of 1; Table 7.2) also show mostly comparable average transmission times for both modules. Notably for the increased buffer sizes, the CFR is now almost always 100 %, except for the 1 Gbps tests with the 1 MiB buffer size. The fastest average transmission time with a CFR of 100 % is achieved with the UDP.c module using an 8 MiB buffer size, completing in 1.34 seconds. Compared to the baseline test results, the average transmission times have hardly changed.

When testing pydiode with a redundancy of 2, the results in Table 7.3 show largely comparable average transmission times for both modules. As in the baseline tests, a CFR of 100 % is consistently achieved. Only with a buffer size of 1 MiB do the average transmission times match those observed during baseline testing; when the buffer size is increased, the average transmission times increase as well.

## 7.2. THE MTU VALUES OF THE DATA DIODE PROXIES

All test results discussed thus far are obtained with the MTU of both NICs set to the default value of 1500 bytes. For the test setup, the maximum MTU ( $S_{MTU,max}$ ) appears to be 9194 bytes. Although this is significantly lower than the theoretical maximum of 65,507 bytes, it

is still more than six times the default value.

To determine the  $S_{MTU,max}$  for the test setup, a manual search using a binary approach was conducted. The lower bound was set at 9000 bytes, as discussed in Section 2.3.2, while the upper bound was arbitrarily set at 16,384 bytes. Listing 7.2 illustrates the type of commands applied during this procedure, including the error message returned by the system when the specified  $S_{MTU,max}$  was not supported. This procedure identified 9194 bytes as the value of  $S_{MTU,max}$  for this configuration.

```
# The first command returns an error message, the second command is accepted.
$ sudo ip link set dev enp1s0 mtu 9195
Error: mtu greater than device maximum.
$ sudo ip link set dev enp1s0 mtu 9194
$
```

Listing 7.2: Searching for the maximum MTU value for a NIC.

```
# Setting MTU for both NICs to the maximum supported value of the test configuration.
sudo ip netns exec sender_ns ip link set dev enp1s0 mtu 9194
sudo ip netns exec receiver_ns ip link set dev enp2s0 mtu 9194
```

Listing 7.3: Bash script used to set maximum MTU value for both NICs.

The next test results explore the effect of increasing the MTU value to 9194 bytes, in combination with varying UDP receiver socket buffer sizes.

Recv Socket Buffer	Netcat	default UDP.c		dd_udp.c	
Size	Bitrate	CFR	Avg. Duration	CFR	Avg. Duration
Default	500 Mbps	098.3 %	03.05 s	098.8 %	03.06 s
Default	750 Mbps	095.5 %	02.38 s	093.3 %	02.38 s
Default	1 Gbps	087.7 %	02.06 s	087.3 %	02.04 s
1 MiB	500 Mbps	097.9 %	03.06 s	099.6 %	03.05 s
1 MiB	750 Mbps	095.1 %	02.40 s	095.9 %	02.40 s
1 MiB	1 Gbps	098.5 %	02.05 s	099.1 %	02.04 s
2 MiB	500 Mbps	095.9 %	03.17 s	098.2 %	03.07 s
2 MiB	750 Mbps	093.7 %	02.43 s	095.1 %	02.39 s
2 MiB	1 Gbps	096.2 %	02.07 s	098.6 %	02.04 s
4 MiB	500 Mbps	095.8 %	03.19 s	096.6 %	03.10 s
4 MiB	750 Mbps	091.1 %	02.52 s	095.5 %	02.41 s
4 MiB	1 Gbps	097.0 %	02.08 s	099.0 %	02.04 s
8 MiB	500 Mbps	099.5 %	03.04 s	098.1 %	03.09 s
8 MiB	750 Mbps	096.5 %	02.37 s	095.4 %	02.40 s
8 MiB	1 Gbps	099.9 %	02.03 s	098.3 %	02.09 s
16 MiB	500 Mbps	099.6 %	03.06 s	099.4 %	03.07 s
16 MiB	750 Mbps	097.8 %	02.39 s	097.9 %	02.42 s
16 MiB	1 Gbps	099.8 %	02.03 s	099.4 %	02.05 s

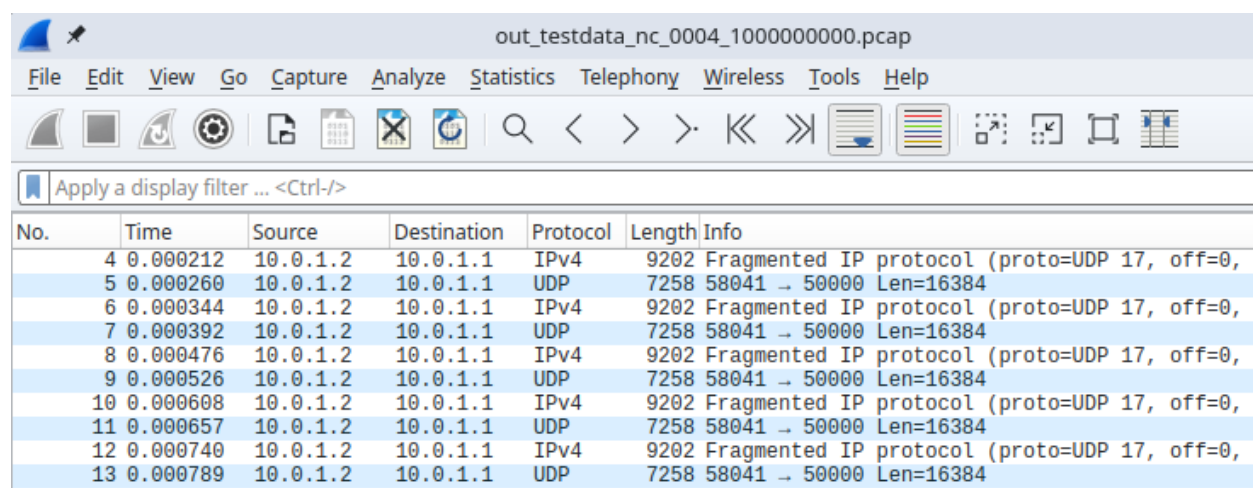
Table 7.4: Adjusted receiver socket buffers and MTU=9194: Netcat test results.

Recv Socket Buffer	pydiode	default UDP.c		dd_udp.c	
Size	Bitrate	CFR	Avg. Duration	CFR	Avg. Duration
Default	500 Mbps	099.5 %	02.27 s	099.7 %	02.30 s
Default	750 Mbps	081.9 %	01.56 s	080.8 %	01.58 s
Default	1 Gbps	035.9 %	01.32 s	034.5 %	01.34 s
1 MiB	500 Mbps	100.0 %	02.25 s	100.0 %	02.26 s
1 MiB	750 Mbps	100.0 %	01.53 s	100.0 %	01.58 s
1 MiB	1 Gbps	099.9 %	01.35 s	099.8 %	01.34 s
2 MiB	500 Mbps	100.0 %	02.23 s	100.0 %	02.26 s
2 MiB	750 Mbps	100.0 %	01.57 s	100.0 %	01.58 s
2 MiB	1 Gbps	100.0 %	01.35 s	100.0 %	01.37 s
4 MiB	500 Mbps	100.0 %	02.30 s	100.0 %	02.31 s
4 MiB	750 Mbps	100.0 %	01.58 s	100.0 %	01.60 s
4 MiB	1 Gbps	100.0 %	01.36 s	100.0 %	<b>01.34 s</b>
8 MiB	500 Mbps	100.0 %	02.32 s	100.0 %	02.29 s
8 MiB	750 Mbps	100.0 %	01.61 s	100.0 %	01.59 s
8 MiB	1 Gbps	100.0 %	01.37 s	100.0 %	01.36 s
16 MiB	500 Mbps	100.0 %	02.29 s	100.0 %	02.28 s
16 MiB	750 Mbps	100.0 %	01.57 s	100.0 %	01.57 s
16 MiB	1 Gbps	100.0 %	01.37 s	100.0 %	01.36 s

Table 7.5: Adjusted receiver socket buffers and MTU=9194: pydiode test results with redundancy=1.

Recv Socket Buffer	pydiode	default UDP.c		dd_udp.c	
Size	Bitrate	CFR	Avg. Duration	CFR	Avg. Duration
Default	500 Mbps	100.0 %	04.37 s	100.0 %	04.31 s
Default	750 Mbps	100.0 %	03.06 s	100.0 %	03.10 s
Default	1 Gbps	100.0 %	02.34 s	100.0 %	02.28 s
1 MiB	500 Mbps	100.0 %	04.36 s	100.0 %	04.33 s
1 MiB	750 Mbps	100.0 %	03.16 s	100.0 %	03.06 s
1 MiB	1 Gbps	100.0 %	02.35 s	100.0 %	02.33 s
2 MiB	500 Mbps	100.0 %	04.36 s	100.0 %	04.32 s
2 MiB	750 Mbps	100.0 %	03.22 s	100.0 %	03.18 s
2 MiB	1 Gbps	100.0 %	02.43 s	100.0 %	02.41 s
4 MiB	500 Mbps	100.0 %	04.42 s	100.0 %	04.30 s
4 MiB	750 Mbps	100.0 %	03.20 s	100.0 %	03.28 s
4 MiB	1 Gbps	100.0 %	02.40 s	100.0 %	02.37 s
8 MiB	500 Mbps	100.0 %	04.38 s	100.0 %	04.29 s
8 MiB	750 Mbps	100.0 %	03.26 s	100.0 %	02.96 s
8 MiB	1 Gbps	100.0 %	02.42 s	100.0 %	02.25 s
16 MiB	500 Mbps	100.0 %	04.34 s	100.0 %	04.25 s
16 MiB	750 Mbps	100.0 %	03.21 s	100.0 %	02.97 s
16 MiB	1 Gbps	100.0 %	02.43 s	100.0 %	<b>02.23 s</b>

Table 7.6: Adjusted receiver socket buffers and MTU=9194: pydiode test results with redundancy=2.



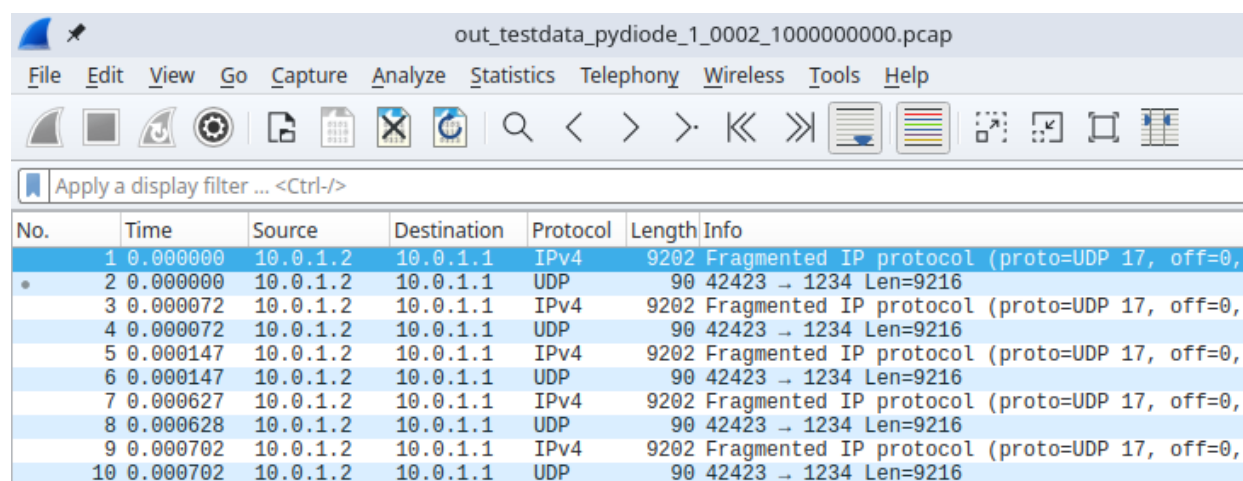
out\_testdata\_nc\_0004\_1000000000.pcap

File Edit View Go Capture Analyze Statistics Telephony Wireless Tools Help

Apply a display filter ... <Ctrl-/>

No.	Time	Source	Destination	Protocol	Length	Info
4	0.000212	10.0.1.2	10.0.1.1	IPv4	9202	Fragmented IP protocol (proto=UDP 17, off=0, ...)
5	0.000260	10.0.1.2	10.0.1.1	UDP	7258	58041 → 50000 Len=16384
6	0.000344	10.0.1.2	10.0.1.1	IPv4	9202	Fragmented IP protocol (proto=UDP 17, off=0, ...)
7	0.000392	10.0.1.2	10.0.1.1	UDP	7258	58041 → 50000 Len=16384
8	0.000476	10.0.1.2	10.0.1.1	IPv4	9202	Fragmented IP protocol (proto=UDP 17, off=0, ...)
9	0.000526	10.0.1.2	10.0.1.1	UDP	7258	58041 → 50000 Len=16384
10	0.000608	10.0.1.2	10.0.1.1	IPv4	9202	Fragmented IP protocol (proto=UDP 17, off=0, ...)
11	0.000657	10.0.1.2	10.0.1.1	UDP	7258	58041 → 50000 Len=16384
12	0.000740	10.0.1.2	10.0.1.1	IPv4	9202	Fragmented IP protocol (proto=UDP 17, off=0, ...)
13	0.000789	10.0.1.2	10.0.1.1	UDP	7258	58041 → 50000 Len=16384

Figure 7.1: Fragmentation of UDP datagrams sent by Netcat with MTU=9194 bytes.



out\_testdata\_pydiode\_1\_0002\_1000000000.pcap

File Edit View Go Capture Analyze Statistics Telephony Wireless Tools Help

Apply a display filter ... <Ctrl-/>

No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000	10.0.1.2	10.0.1.1	IPv4	9202	Fragmented IP protocol (proto=UDP 17, off=0, ...)
2	0.000000	10.0.1.2	10.0.1.1	UDP	90	42423 → 1234 Len=9216
3	0.000072	10.0.1.2	10.0.1.1	IPv4	9202	Fragmented IP protocol (proto=UDP 17, off=0, ...)
4	0.000072	10.0.1.2	10.0.1.1	UDP	90	42423 → 1234 Len=9216
5	0.000147	10.0.1.2	10.0.1.1	IPv4	9202	Fragmented IP protocol (proto=UDP 17, off=0, ...)
6	0.000147	10.0.1.2	10.0.1.1	UDP	90	42423 → 1234 Len=9216
7	0.000627	10.0.1.2	10.0.1.1	IPv4	9202	Fragmented IP protocol (proto=UDP 17, off=0, ...)
8	0.000628	10.0.1.2	10.0.1.1	UDP	90	42423 → 1234 Len=9216
9	0.000702	10.0.1.2	10.0.1.1	IPv4	9202	Fragmented IP protocol (proto=UDP 17, off=0, ...)
10	0.000702	10.0.1.2	10.0.1.1	UDP	90	42423 → 1234 Len=9216

Figure 7.2: Fragmentation of UDP datagrams sent by pydiode with redundancy=1 and MTU=9194 bytes.

Figure 7.1 shows the IP packet fragmentation during testing with Netcat. The fragmentation of data send with pydiode is shown in Figure 7.2. These figures also indicate that Netcat sends data segments of 16 KiB, while pydiode sends data segments 9216 bytes.

### 7.2.1. OBSERVATIONS

Table 7.4 presents the Netcat test results. Compared to the baseline results, there are almost no differences in average transmission times. In nearly all cases, the CFR benefits from buffer sizes larger than the default 208 KiB and from the increased MTU value. However, a CFR of 100 % is never achieved. Compared to the Netcat results discussed in Section 7.1.1, no significant differences in CFR or average transmission times can be observed, regardless of the module used. It is noteworthy that at 1 Gbps, the CFRs observed with the default buffer size are much lower (87.7 and 87.3 %) for both modules than in the other 1 Gbps measurements ( $\geq 96.2$  %).

The tests with pydiode (redundancy of 1; Table 7.5) also show mostly comparable average transmission times for both modules. Notably for the increased buffer sizes, the CFR is now almost always 100 %, except for the 1 Gbps tests for the buffer size of 1 MiB. The



fastest average transmission time with a CFR of 100 % is achieved with the `dd_udp.c` module, using a 4 MiB buffer size, completing in 1.34 seconds. In comparison with the results in Section 7.1.1, there is little difference in the average transmission times.

The test results for `pydiode` with `redundancy=2` in Table 7.6 present a somewhat different picture. As described in Section 7.1.1, the CFR remains 100 % in all cases. In this situation, `pydiode` does not appear to benefit from the maximum MTU value; the average transmission times are in most cases equal to or higher than the values found in Section 7.1.1. The `dd_udp.c` module seems to perform slightly better. But only the results for buffer sizes of 8 and 16 MiB with the `dd_udp.c` module show some correspondence with the baseline test results.

## 7.3. ADJUSTING DATA SEGMENTS TO THE MAXIMUM MTU VALUE

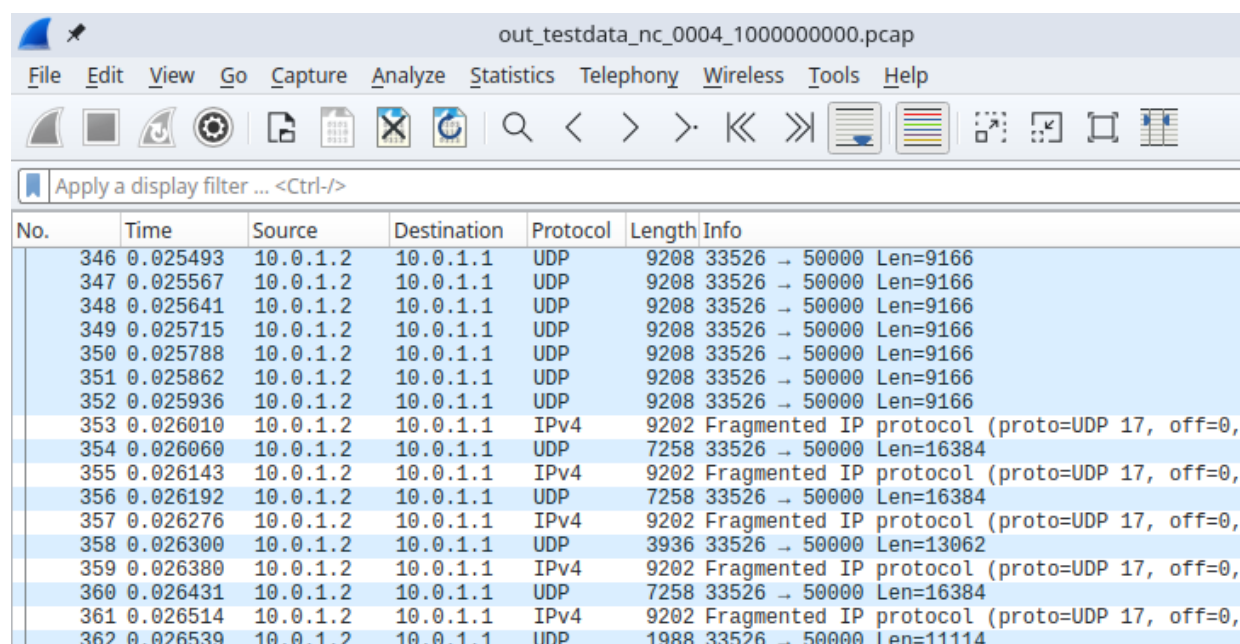
The tests in the previous section use the maximum possible MTU of the test setup. However, Netcat transmits data segments of 16384 bytes, while `pydiode` (version 0.0.1) sends segments of 9216 bytes; as evidenced by Figures 7.1 and 7.2. Both values exceed the MTU limit of 9194 bytes, as imposed by the test setup.

For a data segment to fit within a single Ethernet frame utilizing the maximum MTU, Equation C.2 implies that it must satisfy the condition  $S_{\text{segment}} \leq S_{\text{MTU,max}} - 28$ . This corresponds to a maximum data segment size of 9166 bytes per Ethernet frame. The following test results examine the impact of limiting the data segment size to 9166 bytes in combination with varying UDP receiver socket buffer sizes.

### 7.3.1. ADJUSTING THE DATA SEGMENT SIZE

#### NETCAT

It was not possible to reliably transmit data segments of 9166 bytes with Netcat in a consistent manner. Although transmission initially starts with data segments of 9166 bytes, after some time Netcat reverts to sending data segments of 16 KiB. This is shown in Figure 7.3. The issue occurs mainly at a transmission speed of 1 Gbps) and to a lesser extent at 750 Mbps. At 500 Mbps, the problem does not occur. Attempts were made using the *regulator.py* script (as mentioned in the Research chapter), the Linux *dd* command, and the *-I* and *-O* options as offered by the OpenBSD version of Netcat. All of these produced the same result. Therefore, these Netcat results are not included in this study.



No.	Time	Source	Destination	Protocol	Length	Info
346	0.025493	10.0.1.2	10.0.1.1	UDP	9208	33526 → 50000 Len=9166
347	0.025567	10.0.1.2	10.0.1.1	UDP	9208	33526 → 50000 Len=9166
348	0.025641	10.0.1.2	10.0.1.1	UDP	9208	33526 → 50000 Len=9166
349	0.025715	10.0.1.2	10.0.1.1	UDP	9208	33526 → 50000 Len=9166
350	0.025788	10.0.1.2	10.0.1.1	UDP	9208	33526 → 50000 Len=9166
351	0.025862	10.0.1.2	10.0.1.1	UDP	9208	33526 → 50000 Len=9166
352	0.025936	10.0.1.2	10.0.1.1	UDP	9208	33526 → 50000 Len=9166
353	0.026010	10.0.1.2	10.0.1.1	IPv4	9202	Fragmented IP protocol (proto=UDP 17, off=0, Len=16384
354	0.026060	10.0.1.2	10.0.1.1	UDP	7258	33526 → 50000 Len=16384
355	0.026143	10.0.1.2	10.0.1.1	IPv4	9202	Fragmented IP protocol (proto=UDP 17, off=0, Len=16384
356	0.026192	10.0.1.2	10.0.1.1	UDP	7258	33526 → 50000 Len=16384
357	0.026276	10.0.1.2	10.0.1.1	IPv4	9202	Fragmented IP protocol (proto=UDP 17, off=0, Len=16384
358	0.026300	10.0.1.2	10.0.1.1	UDP	3936	33526 → 50000 Len=13062
359	0.026380	10.0.1.2	10.0.1.1	IPv4	9202	Fragmented IP protocol (proto=UDP 17, off=0, Len=16384
360	0.026431	10.0.1.2	10.0.1.1	UDP	7258	33526 → 50000 Len=16384
361	0.026514	10.0.1.2	10.0.1.1	IPv4	9202	Fragmented IP protocol (proto=UDP 17, off=0, Len=16384
362	0.026539	10.0.1.2	10.0.1.1	UDP	1988	33526 → 50000 Len=11114

Figure 7.3: Problem with UDP datagrams sent by Netcat; MTU=9194 bytes; segment size of 9166 bytes.

## PYDIODE

Pydiode (v0.0.1) does not provide a setting to configure the size of data segments. However, the code shown in Listing 7.4 can be modified. For the purposes of these tests, the value of `UDP_MAX_BYTES` in the `common.py` module was changed from 9216 to 9166.

```
import logging
import struct

# How much data will fit in each packet we send?
# Experimentally, this is the maximum UDP payload I can send on macOS.
UDP_MAX_BYTES = 9216
```

Listing 7.4: Start fragment of the 'common.py' module (pydiode 0.0.1).

## 7.3.2. OBSERVATIONS

As in the previous sections, the tests with pydiode (redundancy of 1; Table 7.7) show largely comparable average transmission times for both modules. Only minor differences are observed compared to the results in the previous sections, as well as to the baseline test results. The fastest average transmission time with a CFR of 100 % is achieved by both the `UDP.c` module with an 8 MiB buffer and the `dd_udp.c` module with a 1 MiB receiver buffer, each completing in 1.32 seconds.

The test results for pydiode with redundancy=2 (Table 7.8) show the same tendency as noted in the previous sections. The CFR remains 100 % in all cases, but pydiode does not appear to benefit from the maximum MTU value, nor from aligning the data segment size to this maximum. The average transmission times are comparable to the values found in the previous sections, whereas the baseline test results demonstrate better performance than those shown in Table 7.7. The `dd_udp.c` module appears to perform slightly better than the `UDP.c` module. The fastest average transmission time with a CFR of 100 % is achieved by the `dd_udp.c` module using a 1 MiB buffer size, completing in 2.23 seconds.

Recv Socket Buffer	pydiode	default UDP.c		dd_udp.c	
Size	Bitrate	CFR	Avg. Duration	CFR	Avg. Duration
Default	500 Mbps	099.8 %	02.30 s	099.8 %	02.25 s
Default	750 Mbps	075.5 %	01.61 s	075.6 %	01.55 s
Default	1 Gbps	042.1 %	01.32 s	041.7 %	01.31 s
1 MiB	500 Mbps	100.0 %	02.31 s	100.0 %	02.24 s
1 MiB	750 Mbps	100.0 %	01.60 s	100.0 %	01.54 s
1 MiB	1 Gbps	099.9 %	01.34 s	100.0 %	<b>01.32 s</b>
2 MiB	500 Mbps	100.0 %	02.28 s	100.0 %	02.25 s
2 MiB	750 Mbps	100.0 %	01.55 s	100.0 %	01.56 s
2 MiB	1 Gbps	100.0 %	01.36 s	100.0 %	01.33 s
4 MiB	500 Mbps	100.0 %	02.42 s	100.0 %	02.30 s
4 MiB	750 Mbps	100.0 %	01.71 s	100.0 %	01.62 s
4 MiB	1 Gbps	100.0 %	01.39 s	100.0 %	01.33 s
8 MiB	500 Mbps	100.0 %	02.23 s	100.0 %	02.28 s
8 MiB	750 Mbps	100.0 %	01.53 s	100.0 %	01.58 s
8 MiB	1 Gbps	100.0 %	<b>01.32 s</b>	100.0 %	01.33 s
16 MiB	500 Mbps	100.0 %	02.26 s	100.0 %	02.31 s
16 MiB	750 Mbps	100.0 %	01.55 s	100.0 %	01.65 s
16 MiB	1 Gbps	100.0 %	01.33 s	100.0 %	01.38 s

Table 7.7: Adjusted rcv socket buffers; MTU=9194: pydiode with redundancy=1; 9166 byte data segments.

Recv Socket Buffer	pydiode	default UDP.c		dd_udp.c	
Size	Bitrate	CFR	Avg. Duration	CFR	Avg. Duration
Default	500 Mbps	100.0 %	04.35 s	100.0 %	04.28 s
Default	750 Mbps	100.0 %	03.02 s	100.0 %	03.00 s
Default	1 Gbps	100.0 %	02.26 s	100.0 %	02.26 s
1 MiB	500 Mbps	100.0 %	04.33 s	100.0 %	04.29 s
1 MiB	750 Mbps	100.0 %	03.00 s	100.0 %	02.97 s
1 MiB	1 Gbps	100.0 %	02.28 s	100.0 %	<b>02.23 s</b>
2 MiB	500 Mbps	100.0 %	04.30 s	100.0 %	04.30 s
2 MiB	750 Mbps	100.0 %	02.99 s	100.0 %	02.96 s
2 MiB	1 Gbps	100.0 %	02.25 s	100.0 %	02.24 s
4 MiB	500 Mbps	100.0 %	04.34 s	100.0 %	04.28 s
4 MiB	750 Mbps	100.0 %	03.12 s	100.0 %	02.95 s
4 MiB	1 Gbps	100.0 %	02.29 s	100.0 %	02.24 s
8 MiB	500 Mbps	100.0 %	04.45 s	100.0 %	04.27 s
8 MiB	750 Mbps	100.0 %	03.21 s	100.0 %	02.96 s
8 MiB	1 Gbps	100.0 %	02.34 s	100.0 %	02.26 s
16 MiB	500 Mbps	100.0 %	04.42 s	100.0 %	04.32 s
16 MiB	750 Mbps	100.0 %	03.23 s	100.0 %	03.06 s
16 MiB	1 Gbps	100.0 %	02.45 s	100.0 %	02.27 s

Table 7.8: Adjusted rcv socket buffers; MTU=9194: pydiode with redundancy=2; 9166 byte data segments.

# 8

## DISCUSSION

This chapter discusses the choices made during this study, addresses potential limitations and points of uncertainty, and validates the results obtained.

### 8.1. COMPARING BASELINE RESULTS TO PREVIOUS RESEARCH

This study builds on the work of Story [Story, 2023b], published in September 2023. To validate the test setup used and the values obtained during the baseline tests, these values are compared with the results published by Story. As in Story’s methodology, for each combination of configuration and transmission speed, a file was sent through the data diode hardware 1,000 times. However, the controlling Python test scripts<sup>1</sup> were specifically developed for this research.

In his paper, Story mentions using Ubuntu 22.04 on his workstation, but the kernel version is not specified. It is possible that certain low-level kernel changes were already implemented before the tests conducted in this study, which uses kernel version 6.8.0-58-generic. One such change concerns the default process scheduler. Since October 2023, the EEVDF process scheduler<sup>2</sup> has been available in Linux kernel 6.6<sup>3</sup>. Until then, the CFS scheduler was used by default. Kubuntu 24.04.2, which includes kernel version 6.8, uses the new EEVDF scheduler as its default.

#### NETCAT

Table 8.1 shows the baseline test results obtained with Netcat. For comparison, the results from Story have been included. The average transmission times are nearly identical for the three situations shown. The CFR values found by Story are generally higher; in this study, a CFR of 100 % was never observed for Netcat.

---

<sup>1</sup><https://github.com/Gert-Jan1966/DataDiodesAndPacketLoss/tree/main/Testscripts>

<sup>2</sup><https://docs.kernel.org/scheduler/sched-eevdf.html>

<sup>3</sup><https://unix.stackexchange.com/questions/127110/which-process-scheduler-is-my-linux-system-using/127113#127113>

Netcat	default UDP.c		dd_udp.c		Story	
Bitrate	CFR	Avg. Duration	CFR	Avg. Duration	CFR	Avg. Duration
100 Mbps	095.0 %	11.03 s	095.6 %	11.04 s	100.0 %	11.02 s
250 Mbps	094.6 %	05.05 s	093.9 %	05.06 s	099.9 %	05.03 s
500 Mbps	097.3 %	03.04 s	096.8 %	03.04 s	098.4 %	03.03 s
750 Mbps	093.3 %	02.37 s	094.3 %	02.39 s	096.7 %	02.36 s
1 Gbps	097.6 %	02.07 s	095.6 %	02.08 s	098.6 %	02.07 s

Table 8.1: Validating baseline Netcat test results.

pydiode		default UDP.c		dd_udp.c		Story	
Bitrate	Redundancy	CFR	Avg. Duration	CFR	Avg. Duration	CFR	Avg. Duration
100 Mbps	1	100.0 %	10.43 s	100.0 %	10.35 s	099.8 %	10.35 s
250 Mbps	1	100.0 %	04.28 s	100.0 %	04.24 s	099.1 %	04.25 s
500 Mbps	1	099.9 %	02.21 s	100.0 %	02.19 s	096.1 %	02.18 s
750 Mbps	1	091.3 %	01.56 s	089.3 %	01.53 s	086.7 %	01.52 s
1 Gbps	1	060.1 %	01.35 s	062.5 %	01.33 s	081.8 %	01.13 s
100 Mbps	2	100.0 %	20.44 s	100.0 %	20.44 s	100.0 %	20.49 s
250 Mbps	2	100.0 %	08.31 s	100.0 %	08.31 s	100.0 %	08.36 s
500 Mbps	2	100.0 %	04.35 s	100.0 %	04.32 s	100.0 %	04.28 s
750 Mbps	2	100.0 %	03.01 s	100.0 %	02.96 s	100.0 %	02.94 s
1 Gbps	2	100.0 %	<b>02.18 s</b>	100.0 %	<b>02.18 s</b>	100.0 %	<b>02.17 s</b>

Table 8.2: Validating baseline pydiode test results.

## PYDIODE

The baseline results and the reference results for pydiode are presented in Table 8.2. In most cases, the average transmission times are consistent, except at a speed of 1 Gbps with redundancy=1. There, the observed transmission times (for both kernel modules) lag 0.2 seconds behind the value measured by Story.

In terms of CFR, when redundancy=1, the setup used in this study appears to perform better than Story’s setup. But this only holds for the lower transmission speeds, making the result less significant. When tested at 1 Gbps, the CFRs observed for both modules are much lower (60.1 % and 62.5 %) compared to Story’s reported result of 81.8 %. This study also confirms that the CFR is 100 % in all cases when the redundancy equals 2.

## CONCLUSION

The findings of this research closely align with those reported by Story. Lower bitrates result in more reliable outcomes, and using pydiode with a redundancy of 2 ensures that all transfers succeed. Furthermore, the change in process scheduler from CFS to EEVDF does not appear to produce noticeable differences. These results suggest that the test setup—both the hardware and the test scripting—provides a sufficiently reliable basis for the experiments.

## 8.2. USING A NUC AS A DATA DIODE TESTING PLATFORM

The test platform used in this study is a single GMKtec NucBox M5 Plus; as shown in Figure 4.2. It is built around an AMD Ryzen 7 5825U CPU and offers two 2.5 Gbps RJ45 network ports, WiFi 6E, 64 GB of RAM, and a 1 TB SSD for storage. This NUC functions as both the TCP Sending Proxy and the TCP Receiving Proxy shown in Figures 2.5 and 3.2, which simplifies the measurement of performance and packet loss. However, this configuration also has certain drawbacks.

### 8.2.1. KUBUNTU OS

The NUC operates on Kubuntu 24.04.2 LTS, a modern, feature-rich, and user-friendly operating system. However, this also entails that numerous background processes are active, many of which are unnecessary for the operation of a data diode setup.

To minimize background processing, the WiFi connection is disabled during testing. Additionally, services such as Baloo (the KDE file indexer) and Bluetooth are also disabled. Although a number of background processes have been disabled, it remains possible that additional processes, which are unnecessary for the data diode setup, are still active and consuming system resources.

### 8.2.2. SYSTEM LOAD LIMITATIONS

During preliminary testing and the conduct of experiments, some limitations of the test setup were observed.

#### TCPDUMP

As discussed in Section 6.5, the use of tcpdump proved impractical during performance testing. Although the NUC is capable of running tcpdump for both the sending and receiving processes simultaneously, the CFR decreased significantly across all test cases, indicating a notable impact on data transmission reliability. Support for tcpdump in the test scripts was retained to allow the investigation of edge cases. This is illustrated in Figure 7.3 in Section 7.3.1, which demonstrates the issue whereby Netcat does not consistently respect the specified data segment size.

#### SYSTEM PERFORMANCE DEGRADATION

Another practical issue arose during prolonged testing of the setup. After approximately 30,000 executed test cases (trials), system performance occasionally degraded: average transmission times increased, and the system became less responsive. This effect was observed regardless of which kernel module was tested. However, when specific stress tests were conducted to reproduce this behavior, it did not occur.

To mitigate this, the tests were executed in smaller batches, after which the NUC was rebooted. This approach helped maintain system responsiveness and ensured that transmission times remained stable throughout the experiments.

## 8.3. DD\_UDP.C AS A LOADABLE KERNEL MODULE

With the implementation of the alternative dd\_udp.c module, one might question whether Research Question 1 (*“How can the Linux UDP.c kernel module be implemented or modified with minimizing packet loss during data transfer in mind?”*) has been adequately answered.

The challenges of a complete reimplementaion are described in Section 5.1.1, while the realized implementation is discussed in Section 5.2. Although it is not denied that a full reimplementaion might achieve greater efficiency, there are additional reasons to prefer the implemented `dd_udp.c` solution.

The UDP.c module can be characterized as a collection of functions that hook into the Linux networking stack to implement the User Datagram Protocol (UDP) for IPv4. These functions are called at specific points in the networking stack to process UDP datagrams, handle socket operations, and interface with the Internet Layer.

This makes it easier to reimplement or bypass parts of the functionality with a Loadable Kernel Module (LKM), which is exactly what has been done in the implementation of `dd_udp.c`. However, this requires the use of Netfilter hooks in order to access the datagrams. Furthermore, the LKM approach enables building and testing the kernel module without requiring repeated reboots of the NUC or workstation to insert the updated module into the kernel.

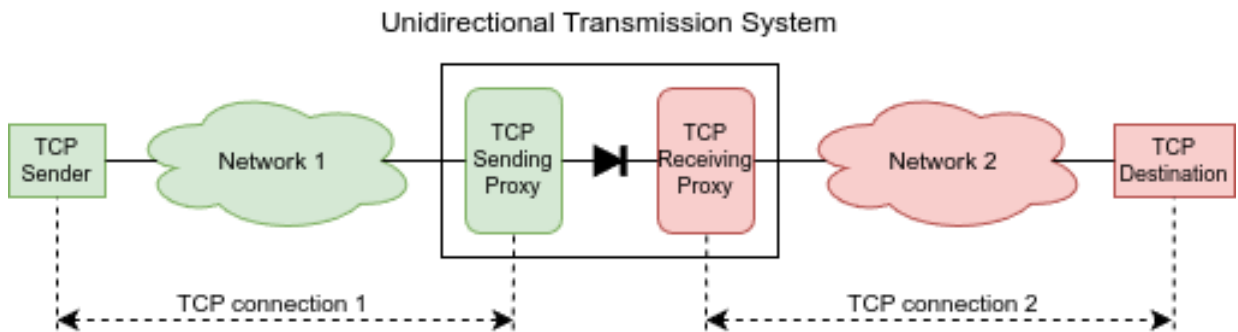


Figure 8.1: Basic data diode architecture with TCP proxies (adopted from [Kim and Min, 2016]).

Finally, the `dd_udp.c` module makes it possible to continue using the standard UDP.c module within the default Linux network namespace. This provides advantages when applied in the Sending and Receiving Proxies of a data diode. The Sending Proxy disassembles the incoming protocol from the TCP Sender and forwards the data to the Receiving Proxy using UDP. The Receiving Proxy accepts the UDP data and reconstructs an outgoing protocol to deliver the data to the TCP destination. With the `dd_udp.c` module, it is therefore possible—if desired—to use standard UDP in the Sending Proxy to receive UDP traffic and in the Receiving Proxy to forward data via standard UDP to the final destination, while the transfer between the Proxies is handled by the `dd_udp.c` module.

# 9

## CONCLUSION AND FUTURE WORK

This chapter addresses the research questions formulated in this study and outlines recommendations for future research.

### 9.1. CONTEXT

The research was conducted within the context of a simplified data diode setup, using a 1 Gbit file for testing, with the maximum transmission speed limited to 1 Gbps by the applied hardware.

### 9.2. CONCLUSIONS

#### RESEARCH QUESTION 1

*"How can the Linux UDPc kernel module be implemented or modified with minimizing packet loss during data transfer in mind?"*

Due to its critical role, deep integration, and direct compilation into the Linux kernel, the UDPc module cannot be replaced on a running system without disrupting compatibility, performance optimizations, and security mechanisms, making substitution highly challenging; instead, the new `dd_udp.c` module is introduced not as a direct replacement, but as a custom overlay designed to implement a UDP-based data diode with enhanced features such as packet loss and corruption tracking for testing, as well as awareness of hardware checksum offloading.

Although the results did not show a substantial improvement in CFR or a measurable reduction in transmission times, the implementation itself demonstrates the feasibility of selectively re-engineering the standard UDPc module. This contributes to a deeper understanding of which functionalities are essential in data diode contexts and provides a foundation for future optimizations.

#### RESEARCH QUESTION 2

*"What are relevant measuring points for monitoring UDP data transfer, to facilitate enhancements aimed to minimizing packet loss?"*

Identifying usable measurement points within the Linux networking stack was the objective of the second research question. This led to the selection of actual measurement points



based on the Linux `/proc/net/snmp` virtual file, datagram counters in the `dd_udp.c` module, and the virtual `sysfs` filesystem in Linux. These measurement points proved to be valuable for investigating the other research questions. During the tests conducted using these measurement points, it was found that data loss occurred only at the UDP receiver socket buffer.

### RESEARCH QUESTIONS 3, 4, AND 5

*"What is the impact of adjusting the UDP receiver socket buffer size on packet loss during transmission?"*

*"How does changing the MTU to its maximum value influences packet loss during the transmission of UDP datagrams?"*

*"How does aligning data segment sizes with the maximum MTU value affects transmission performance and packet loss?"*

When examining the results of adjustments to the internal parameters of the diode proxies, only increasing the size of the UDP receiver buffer (research question 3) shows a clearly observable impact: `pydiode`, with a redundancy setting of 1, consistently achieves a CFR of 100 % once the buffer size reaches at least 2 MiB.

This effect remains present when the MTU is increased to 9194 bytes (research question 4) and also when the data segment size is subsequently adjusted to 9166 bytes (research question 5). For `Netcat`, some improvement is observed in these situations, but it never reaches a CFR of 100 %. `Pydiode` with a redundancy setting of 2 does not benefit from these parameter changes. As in the baseline tests, the CFR always remains at 100 %, but the average transmission time either remains the same or increases. In cases where the transmission times increase with larger buffer size values, the `dd_udp.c` module shows better performance than the standard `UDP.c` module.

### CONCLUDING REMARKS

The measured CFR values and average throughput show minor performance differences between the two kernel modules, with no module consistently outperforming the other. From a performance and efficiency standpoint, there is no compelling reason to prefer the `dd_udp.c` kernel module over the `UDP.c` kernel module.

Packet loss consistently results from overflow of the UDP receiver socket buffer. There is no packet loss within the data diode hardware. Enlarging the receiver socket buffer can effectively mitigate such loss. This suggests that further reductions in packet loss may be achievable through targeted software and configuration optimizations within the receiver proxy, as well as through enhancements to the data transfer software itself.

If transfer speed is more important than a small risk of data loss, `pydiode` with a redundancy setting of 1 and a UDP receiver socket buffer size greater than 2 MiB is a viable option. If preventing data loss is a higher priority, `pydiode` with a redundancy setting of 2 remains the safer choice.

`Netcat` is not a good option for UDP data transfer with a data diode. The proposed changes did not affect transfer times and while the CFR did increase, a 100 % was never achieved.

### 9.3. FUTURE WORK

Future work could further investigate how the `dd_udp.c` module might be optimized to achieve measurable improvements in CFR and transmission times. This includes exploring alternative approaches to re-engineering the standard UDP.c module, assessing the impact of buffer management, and offloading features, and evaluating the module's performance on different hardware platforms or under higher data rates.

An open question is how the results would differ if the same study were conducted using two separate computers instead of one NUC. It should be noted that the current version of the `dd_udp.c` module is implemented for use on a single computer with two NICs. During module initialization, two network namespaces and two NICs must be specified. This limitation can be addressed by modifying the module's `diode_init()` and `diode_exit()` functions, making it possible to load the revised `dd_udp.c` module on separate Sending and Receiving Proxies. With this modification, an additional security measure is immediately introduced. Since both a network namespace and the associated NIC must be specified during initialization, using a dedicated namespace ensures that the associated NIC is used exclusively for this UDP data traffic.

A limitation of the present study is that the experiments were restricted to files of 1 Gbit (125 MB). Future work should investigate whether the results reported here remain consistent when testing with larger files (250, 500, 750, and 1,000 MB), as larger file sizes may reveal additional effects related to buffering, memory management, or throughput.

Future research could investigate how a similar test setup performs — particularly with respect to packet loss — at higher data transmission rates, such as 2.5, 5, 7.5, and 10 Gbps. This would help to assess the scalability of the current setup and identify potential bottlenecks under more demanding conditions.

An open question is the effect of different scheduling and execution strategies for the sender and receiver processes. In particular, binding processes to specific processor cores (core pinning) could be evaluated. Furthermore, the impact of increasing process priorities on packet loss warrants investigation. Evaluating these strategies under a real-time operating system could further clarify how deterministic scheduling influences reliable data transfer within data diodes.

Future work could investigate the impact of using IPv6 instead of IPv4 on packet loss in a comparable data diode setup. Since IPv6 introduces a different header format and mandates the use of UDP checksums for error detection, this may influence processing overhead and affect packet loss characteristics.

Another limitation of the present study is the reliance on a static test file. An open question is how the data diode hardware, particularly with respect to packet loss, behaves when subjected to continuous data streams. In such scenarios, the CFR metric used in this study can no longer serve as an effective measure for characterizing the performance of the test setup.

Future work could explore mechanisms for adjusting the data transmission rate in real

time when streaming data risks overflowing the buffers of the sending proxy, taking into account the current buffer status or fill level.

In [Honggang, 2013], the relationship between packet loss rate and transmission rate is investigated, and a two-state Markov chain model for characterizing packet loss in unidirectional transmission is derived. This model could potentially be extended—or a new model developed—to calculate the required redundancy for a given data size while ensuring that the probability of transmission failure does not exceed a specified threshold. This would be particularly important for long-running transfers. Using a statistical estimate may be a feasible alternative to collecting data for larger files, given the significant time required for data collection.

Finally, the issue described in Section 6.5.4 and Appendix D—the increasing `SndbufErrors` value when using `pydiode`—deserves further investigation. It is possible that the issue is related to the combination of `pydiode` and the Python `asyncio` library; however, it could also be an imperfection within the Kubuntu Linux environment.

# BIBLIOGRAPHY

- Guy Almes, Sunil Kalidindi, Marija Zekauskas, and Al Morton. *A One-Way Loss Metric for IP Performance Metrics (IPPM)*, 2016. <https://datatracker.ietf.org/doc/rfc7680/>. 10
- P.T.N.T.D. An and T.T. Tin. Design of unidirectional security gateway device for secure data transfer. Proceedings of the 14th National Scientific and Technological Conference on Fundamental Research and Application of Information Technology (FAIR), Ho Chi Minh City, Vietnam, 2021. 9, 13
- Ross D Arnold. Strategies for transporting data between classified and unclassified networks. Technical report, Technical report, Defense Technical Information Center, Fort Belvoir, VA, 2016. 1
- Chirag Bhalodia. *Chirag's Blog: UDP Header*, 2021. <https://www.chiragbhalodia.com/2021/12/udp-header.html>. vi, 6
- Cisco. *What Is Network Segmentation?* <https://www.cisco.com/c/en/us/products/security/what-is-network-segmentation.html>. 1
- Joe Damato. *Monitoring and Tuning the Linux Networking Stack: Receiving Data | Packagecloud Blog*, 2016. <https://blog.packagecloud.io/monitoring-tuning-linux-networking-stack-receiving-data/>. 22
- Joe Damato. *Monitoring and Tuning the Linux Networking Stack: Sending Data | Packagecloud Blog*, 2017. <https://blog.packagecloud.io/monitoring-tuning-linux-networking-stack-sending-data/>. 22, 23, 29, 32
- Miguel Borges de Freitas, Luis Rosa, Tiago Cruz, and Paulo Simões. Sdn-enabled virtual data diode. In *International Workshop on Security and Privacy Requirements Engineering*, pages 102–118. Springer, 2018. 1
- The Hague Security Delta. Understanding the strategic and technical significance of technology for security: The case of data diodes for cybersecurity, 2019. [https://securitydelta.nl/media/com\\_hsd/report/246/document/HSD-Rapport-Data-Diodes.pdf](https://securitydelta.nl/media/com_hsd/report/246/document/HSD-Rapport-Data-Diodes.pdf). 2, 12
- docs.kernel.org. *Introduction to Netlink — The Linux Kernel documentation*, 2025. <https://docs.kernel.org/userspace-api/netlink/intro.html>. 25
- Lin Honggang. Research on packet loss issues in unidirectional transmission. *Journal of computers*, 8(10):2664–2671, 2013. vi, 2, 9, 10, 11, 12, 51
- IETF. *RFC1213: Management Information Base for Network Management of TCP/IP-based internets*, 1991. <https://www.ietf.org/rfc/rfc1213.txt>. 28

- Raj Jain. Error characteristics of fiber distributed data interface (fdi). *IEEE Transactions on Communications*, 38(8):1244–1252, 1990. doi: 10.1109/26.58757. 15
- Jorden Kerkhof. Design for a tcp/ip transparent fpga-based network diode: To what extent is it possible to implement a network diode on an fpga under realistic network environments, using the transmission control protocol? Master’s thesis, TU Delft, 2021. 12
- kernel.org. *sysfs-class-net-statistics*, 2024. <https://www.kernel.org/doc/Documentation/ABI/testing/sysfs-class-net-statistics>. 29
- Muhammad Saleem Khan, Saira Waris, Ihsan Ali, Majid I Khan, and Mohammad Hossein Anisi. Mitigation of packet loss using data rate adaptation scheme in manets. *Mobile Networks and Applications*, 23:1141–1150, 2018. 10
- Dongwook Kim and Byunggil Min. Design of a reliable data diode system. *Journal of the Korea Institute of Information Security & Cryptology*, 26(6):1571–1582, 2016. vi, 9, 10, 12, 47
- Scott Lowe. *Introducing Linux Network Namespaces*, 2013. <https://blog.scottlowe.org/2013/09/04/introducing-linux-network-namespaces/>. 17, 24
- Maxnilz. *Linux Networking Stack tutorial: Receiving Data*, 2025. <https://maxnilz.com/docs/004-network/006-linux-rx-v0/>. vi, 5, 30
- David Murray, Terry Koziniec, Kevin Lee, and Michael Dixon. Large mtus and internet performance. In *2012 IEEE 13th International Conference on High Performance Switching and Routing*, pages 82–87. IEEE, 2012. 7, 15
- NCTV. *Cybersecuritybeeld 2023*, 2023. <https://www.nctv.nl/actueel/nieuws/2023/07/03/cybersecuritybeeld-2023-verwacht-het-onverwachte>. 1
- NMAP.ORG. *Ncat - Netcat for the 21st Century*, 2024. <https://nmap.org/ncat/>. 14
- J Postel. *RFC 768: User Datagram Protocol*, 1980. <https://datatracker.ietf.org/doc/html/rfc768>. 6, 7
- J Postel. *RFC 791: Internet Protocol*, 1981. <https://datatracker.ietf.org/doc/html/rfc791>. 9
- PSF. *A Conceptual Overview of asyncio*, 2025. <https://docs.python.org/3/howto/a-conceptual-overview-of-asyncio.html>. 14
- Anil Rijasinghani. *Computation of the Internet Checksum via Incremental Update*, 1994. <https://www.rfc-editor.org/info/rfc1624>. 7
- Luigi Rizzo. Dummynet and forward error correction. In *1998 USENIX Annual Technical Conference (USENIX ATC 98)*, 1998. 11
- MW Stevens. An implementation of an optical data diode. Technical report, DSTO Electronics and Surveillance Research Laboratory, Adelaide, 1999. vi, 1, 2, 9
- Peter Story. *pydiode*, 2023a. <https://github.com/ClarkuCSCI/pydiode>. 14, 17

- Peter Story. Building an affordable data diode to protect journalists. USENIX Symposium on Usable Privacy and Security (SOUPS) 2023, Anaheim, CA, USA, 2023b. vi, 1, 3, 9, 13, 14, 16, 19, 44
- Tcpdump.Group. Tcpdump & libpcap, 2025. <https://www.tcpdump.org/>. 29
- Mohith Kumar Thummaluru. *Introduction to Netfilter*, 2024. <https://blogs.oracle.com/linux/post/introduction-to-netfilter>. 23
- Linus Torvalds. *linux/net/ipv4/Makefile at master - torvalds/linux*, 2025. <https://github.com/torvalds/linux/blob/master/net/ipv4/Makefile>. 23
- UdpCast. *UDP Cast*, 2016. <https://www.udpcast.linux.lu/>. 14
- Vrolijk. *OSDD Open Source Data Diodes*, 2023. <https://github.com/Vrolijk/OSDD>. vi, 9, 12, 13, 16
- Wikipedia. *Tanenbaum-Torvalds debate*, 2025a. [https://en.wikipedia.org/wiki/Tanenbaum-Torvalds\\_debate](https://en.wikipedia.org/wiki/Tanenbaum-Torvalds_debate). 23
- Wikipedia. *Loadable kernel module*, 2025b. [https://en.wikipedia.org/wiki/Loadable\\_kernel\\_module](https://en.wikipedia.org/wiki/Loadable_kernel_module). 24
- G. Wright and J Scarpati. What are jumbo frames?, 2023. <https://www.techtarget.com/searchnetworking/definition/jumbo-frames>. 8, 15

# A

## CSV TEST RESULT FILES

### A.1. CONTEXT FOR EACH CSV FILE

For each tested configuration, a CSV file is created for every transfer speed.

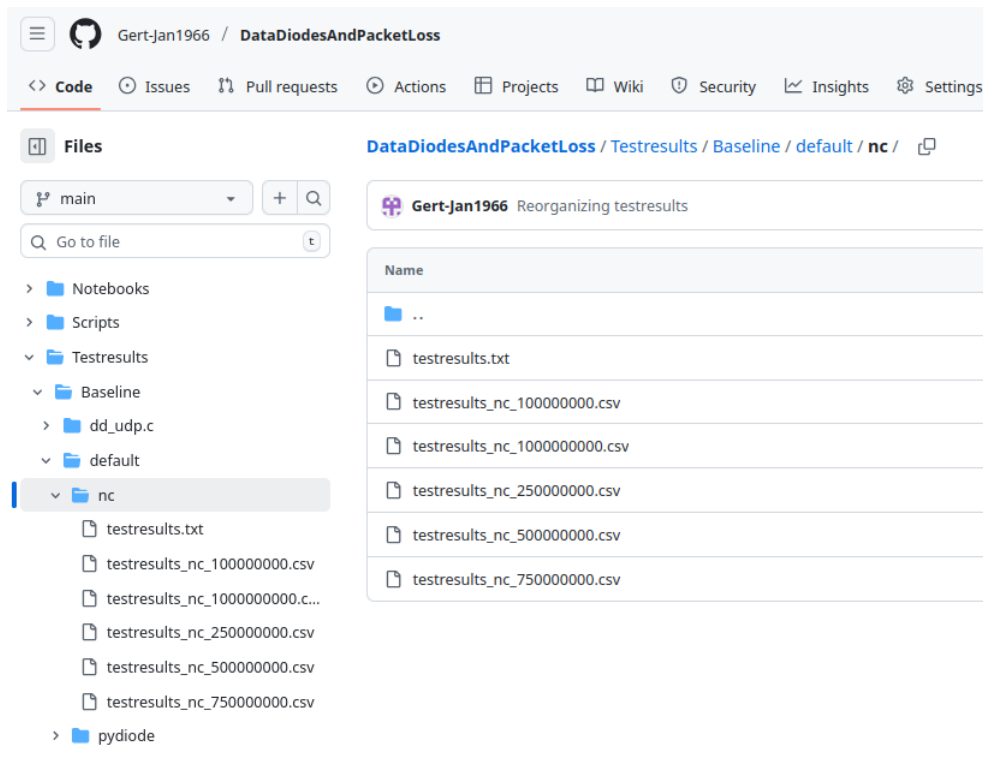


Figure A.1: Example of CSV file directory stored on GitHub

Figure A.1 shows the CSV files resulting from a testrun with the default UDP.c module. Because the test was conducted using five transmission rates, five CSV files are stored in this GitHub directory <sup>1</sup>.

<sup>1</sup><https://github.com/Gert-Jan1966/DataDiodesAndPacketLoss/tree/main/Testresults/Baseline/default/nc>

## A.2. CSV FILE STRUCTURE

Table A.1 details all fields contained in the CSV files generated by the test scripts.

Fieldname	Description
bitrate	Transfer speed [Mbps]
trial	Test case sequence number
sender_rc	Return code of sending program (usually 0)
receiver_rc	Return code of receiving program (usually 0)
Xfer_time	Transfer time of the file [s]
Xfer_success	True if file transfer succeeded, False otherwise
in_file_size	Size of input file [bytes]
out_file_size	Size of output file [bytes]
dtgrms_sent	Number of UDP datagrams intended to be sent for this file (with dd_udp.c module only, 'na' for default UDP.c)
dtgrms_received	Number of UDP datagrams received for this file (with dd_udp.c module only, 'na' for default UDP.c)
dtgrms_corrupt	Number of corrupted UDP datagrams for this file (with dd_udp.c module <i>and</i> if UDP checking is enabled only, 'na' otherwise)
tx_errors	Total number of frames that the system attempted to transmit but failed due to errors for this file, based on Sysfs
tx_dropped	Total number of outbound frames that were dropped before being transmitted by the NIC for this file, based on Sysfs
rx_errors	Total number of frames received by the NIC that were discarded due to errors for this file, based on Sysfs
rx_dropped	Total number of incoming frames that were dropped by the network interface before being handed off to the network stack for this file, based on Sysfs
snmp_OutDatagrams	Number of successfully sent UDP datagrams for this file, based on /proc/net/snmp
snmp_SndbufErrors	Number of data segments failed to be sent due to sending buffer overflow, based on /proc/net/snmp
snmp_InDatagrams	Total number of UDP datagrams delivered to the UDP socket for this file, based on /proc/net/snmp
snmp_InErrors	Total number of incoming UDP datagrams that were received but could not be delivered to the socket due to errors, for this file; based on /proc/net/snmp
snmp_RcvbufErrors	Total number of incoming UDP packets that were dropped because the socket receive buffer was full, for this file; based on /proc/net/snmp

Table A.1: Fields within the CSV files.



# B

## JUPYTER NOTEBOOKS

This appendix describes the Jupyter Notebooks used to analyze the CSV files generated during testing. These notebooks follow a straightforward, step-by-step structure, loading the data, and performing basic processing. The purpose is to provide quick insights, rather than optimized code.

A CSV file is created for each transfer speed within a tested configuration, and the notebooks combine the results from all CSV files of that configuration.

### B.1. NOTEBOOK STRUCTURE: NETCAT - DD\_UDP.C

This is an example stored on GitHub<sup>1</sup>:

#### NOTEBOOK INITIALISATION

```
import pandas as pd

tests1 = pd.read_csv('https://raw.githubusercontent.com/Gert-Jan1966/\
DataDiodesAndPacketLoss/main/Testresults/Baseline/dd_udp.c/nc/\
testresults_nc_100000000.csv')
tests2 = pd.read_csv('https://raw.githubusercontent.com/Gert-Jan1966/\
DataDiodesAndPacketLoss/main/Testresults/Baseline/dd_udp.c/nc/\
testresults_nc_250000000.csv')
tests3 = pd.read_csv('https://raw.githubusercontent.com/Gert-Jan1966/\
DataDiodesAndPacketLoss/main/Testresults/Baseline/dd_udp.c/nc/\
testresults_nc_500000000.csv')
tests4 = pd.read_csv('https://raw.githubusercontent.com/Gert-Jan1966/\
DataDiodesAndPacketLoss/main/Testresults/Baseline/dd_udp.c/nc/\
testresults_nc_750000000.csv')
tests5 = pd.read_csv('https://raw.githubusercontent.com/Gert-Jan1966/\
DataDiodesAndPacketLoss/main/Testresults/Baseline/dd_udp.c/nc/\
testresults_nc_1000000000.csv')
results = pd.concat([tests1, tests2, tests3, tests4, tests5])
results.info()
```

Listing B.1: Notebook initialization.

<sup>1</sup>[https://github.com/Gert-Jan1966/DataDiodesAndPacketLoss/blob/main/Notebooks/baseline/NC-dd\\_udp-tests.ipynb](https://github.com/Gert-Jan1966/DataDiodesAndPacketLoss/blob/main/Notebooks/baseline/NC-dd_udp-tests.ipynb)

During the initialization phase, the *pandas* library, which is employed for statistical analysis, is imported. Subsequently, the test result files (five in this case) are read and concatenated into a single pandas dataframe named *results*. The initialization concludes by outputting summary information about the *results* dataframe to stdout.

## TRANSFER SUCCESS VS TRANSFER FAILURE

```
results['Xfer_success'].value_counts()
```

Listing B.2: Notebook transfer succes vs failure.

Executing this line of code displays the number of successful and failed file transfers on the standard output stream.

```
Xfer_success
True      4762
False     238
Name: count, dtype: int64
```

Listing B.3: Notebook transfer succes vs failure - Output.

## DID WE LOSE FRAMES AT THE LINK LAYER?

```
results.query(' (tx_errors!=0) | (tx_dropped!=0) | (rx_errors!=0) | (rx_dropped!=0) ').value_counts()
```

Listing B.4: Checking for lost frames at the Link Layer.

If frames are lost in the Link Layer, at least one of *tx\_errors*, *tx\_dropped*, *rx\_errors* or *rx\_dropped* indicators should not be zero.

```
Series([], Name: count, dtype: int64)
Name: count, dtype: int64
```

Listing B.5: Lost frames at the Link Layer check - Output.

In this case, none of the indicators  $> 0$ , thus no Ethernet frames were lost.

## DID WE LOSE UDP DATAGRAMS AT THE TRANSPORT LAYER? (NEEDS DD\_UDP.C MODULE)

```
results.query('dtgrms_sent!=dtgrms_received').value_counts()
```

Listing B.6: Checking for lost datagrams in the kernel module.

This line compares the datagrams sent counter to the datagrams received counter. These counters are only available from the `dd_udp.c` kernel module.

```
Series([], Name: count, dtype: int64)
```

Listing B.7: Lost datagrams at the kernel module level check - Output.

This output indicates that no datagrams were lost at the kernel level.

### HOW MANY TRIALS WITH RECEIVER SOCKET BUFFER ERRORS? (LOST SEGMENTS)

```
results.query('snmp_RcvbufErrors!=0').value_counts()
```

Listing B.8: Checking for lost datagrams at the UDP receiver socket buffer.

The result of executing this line of code displays the number of trials in which a buffer overflows of the UDP receiver socket buffer were detected.

```
130    0    0 ...
148    0    0 ...
149    0    0 ...
Name: count, Length: 238, dtype: int64
```

Listing B.9: Lost datagrams at the UDP receiver socket buffer check - Output.

in this case during 238 trials, there were overflows of the UDP receiver socket buffer detected. This is the same amount of trails that failed as indicated in Listing B.3.

### DOES THE NR OF RECEIVER SOCKET INErrors EQUAL THE NR OF RECEIVER BUFFER ERRORS?

```
results.query('snmp_InErrors!=snmp_RcvbufErrors').value_counts()
```

Listing B.10: Checking UDP receiver socket buffer InErrors and RcvbufErrors.

This code actually returns the number of trials in which the InErrors and RcvbufErrors counters differ.

```
Series([], Name: count, dtype: int64)
```

Listing B.11: UDP receiver socket buffer InErrors and RcvbufErrors check - Output.

The empty output indicates that, in all cases, the values of the InErrors and RcvbufErrors counters are identical.

### DO ALL UNSUCCESSFUL TRIALS MATCH ALL TRIALS WITH RECEIVER BUFFER ERRORS?

```
results.query('(snmp_RcvbufErrors!=0)&(Xfer_success==False)').value_counts()
```

Listing B.12: Checking unsuccessful transfers against RcvbufErrors.

The purpose of this code is to verify whether all failed file transfers can be attributed to overflow of the UDP receiver socket buffer.

```

      130      0      0 ...
      148      0      0 ...
      149      0      0 ...
Name: count, Length: 238, dtype: int64

```

Listing B.13: Unsuccessful transfers against RcvbufErrors check - Output.

The number mentioned in the output, 238, matches the number in Listings B.3 and B.9. Thus all failed file transfers can be attributed to UDP datagram receiver socket buffer overflows.

### AND WHAT ABOUT SNMP\_SNDBUFERRORS?

Finally, the number of the SndbufErrors is examined.

```
results.query('(snmp_SndbufErrors_!=0)').value_counts()
```

Listing B.14: Checking SndbufErrors.

```
Series([], Name: count, dtype: int64)
```

Listing B.15: SndbufErrors check - Output.

The output indicates that no errors related to the UDP sending socket buffer were recorded.

# C

## DATA TRANSFER CALCULATIONS

In this thesis, a few calculations concerning data sizes, headers and frames were casually made. This appendix aims to formalize these calculations.

Within the data diode context of this research, utilizing UDP over IPv4 and Ethernet II framing, where the sender NIC always sends data to the same receiver NIC, and no optional IPv4 features are used, the equations as presented in this section can be derived.

By convention,  $N$  is used to represent the number of items.  $S$  denotes a size, typically expressed in bytes

### UDP DATAGRAM SIZE

The size of a UDP datagram can be calculated with:

$$S_{dtgrm,UDP} = S_{hdr,UDP} + S_{segment} \quad \text{or} \\ S_{dtgrm,UDP} = 8 + S_{segment} \quad (C.1)$$

Where:

- $S_{dtgrm,UDP}$  : Size of the UDP datagram.
- $S_{hdr,UDP}$  : Size of the UDP header (8 bytes).
- $S_{segment}$  : Size of the data segment as send by the sending application.

### IP PACKET AND PAYLOAD

The relation between the payload available within an IP packet (MTU) can be described as:

$$S_{payload,pckt} = S_{MTU} - S_{hdr,IPv4} \quad \text{or} \\ S_{payload,pckt} \leq S_{MTU} - 20 \quad (C.2)$$

Where:

- $S_{payload,pckt}$  : Payload size per packet.
- $S_{MTU}$  : Configured MTU size, with  $S_{MTU} \leq 65,535$  bytes.
- $S_{hdr,IPv4}$  : Size of the IPv4 header (20 bytes).

### SIZES OF JUMBO/ETHERNET FRAMES

Since Jumboframes are a kind of Ethernet frames, their size can be calculated using:

$$S_{\text{Jumboframe}} = S_{\text{frame,eth}} = S_{\text{hdr,eth}} + S_{\text{MTU}} + S_{\text{ftr,eth}} \quad \text{or}$$

$$S_{\text{Jumboframe}} = S_{\text{frame,eth}} = S_{\text{MTU}} + 18 \quad (\text{C.3})$$

Where:

- $S_{\text{Jumboframe}}$  : Total size of the Jumboframe to be send.
- $S_{\text{frame,eth}}$  : Total size of the Ethernet frame to be send.
- $S_{\text{hdr,eth}}$  : Size of the Ethernet header (14 bytes).
- $S_{\text{ftr,eth}}$  : Size of the Ethernet footer (4 bytes).

# D

## SNDBUFERRORS > 0 MEASUREMENTS

This appendix details the measurements underlying the SndbufErrors issue described in Section 6.5.4. Table D.1 shows the average increase in SndbufErrors per file sent, while Table D.2 shows the number of files for which an increase in SndbufErrors was observed.

Both tables are compiled from data collected in CSV files generated by the tests conducted for Chapter 7. They include all test series for pydiode with a redundancy setting of 1, using a UDP receiver socket buffer of 4 MiB<sup>1</sup>. In all cases, all sent files were received intact. Since all test cases in this study were executed with a UDP sender socket buffer of 208 KiB, a brief analysis of the test configuration as mentioned above is sufficient.

Kernel Module	default UDP.c			dd_udp.c		
Transfer speed [Mbps]	1000	750	500	1000	750	500
MTU 1500	1509	0	0	1525	0	0
MTU 9194	892	191	0	889	230	0
MTU 9194; Data segment 9166	827	197	0	826	202	0

Table D.1: Average increase in SndbufErrors values per file transferred; pydiode redundancy=1, UDP rcv socket buffer 4MiB.

Kernel Module	default UDP.c			dd_udp.c		
Transfer speed [Mbps]	1000	750	500	1000	750	500
MTU 1500	1000	18	0	1000	3	0
MTU 9194	1000	1000	6	1000	1000	15
MTU 9194; Data segment 9166	1000	1000	11	1000	1000	15

Table D.2: Number of file transfers where SndbufErrors > 0; pydiode redundancy=1, UDP rcv socket buffer 4MiB.

These tables show that most SndbufErrors per file sent occur at a transfer speed of 1 Gbps with an MTU of 1500 bytes. For 750 and 500 Mbps, the average values are zero; however, for a few of the 1,000 files sent at 750 Mbps, the SndbufErrors value did increase.

<sup>1</sup>The Notebook at <https://github.com/Gert-Jan1966/DataDiodesAndPacketLoss/blob/main/Notebooks/SndbufErrors-pydiode-1.ipynb> contains a brief analysis.

Table D.1 also indicates that, at 1 Gbps, the increase in SndbufErrors decreases when the MTU is set to 9,194 bytes, with a further, albeit smaller, decrease when the data segment size is adjusted to 9,166 bytes. Table D.2 shows that, in all these cases, SndbufErrors increased for all sent files.

Furthermore, it is notable that at 750 Mbps the SndbufErrors increase rises significantly when the MTU is set to 9,194 bytes, and this occurs for all sent files. At 500 Mbps, the average SndbufErrors increase remains zero; the few files for which it does occur probably show only a small increase in SndbufErrors.

For the transfer speed of 750 Mbps, some differences between the two kernel modules are observed. However, since the dd\_udp.c module does not manage UDP sockets or their buffer handling, as described in Section 5.2.2, these differences are most likely coincidental.