CHARACTERIZING LATENCIES OF EDGE VIDEO STREAMING

by

Mohamed Balhaj

A thesis submitted to the faculty of
The University of North Carolina at Charlotte
in partial fulfillment of the requirements
for the degree of Master of Science in
Electrical Engineering

Charlotte

2017

Approved by:

_____
Dr. Arun Ravindran

_____
Dr. Tao Han

_____
Dr. Hamed Tabkhi

ABSTRACT

MOHAMED BALHAJ. Characterizing Latencies of Edge Video Streaming. (Under the direction of DR. ARUN RAVINDRAN)

The use of video streaming has been growing rapidly in the recent years and has been utilized in various applications. Many of these applications are latency sensitive, but the latency requirements varies largely from one application to another. The high quality videos captured by cameras are quite large in size, so they are encoded to reduce the video size to achieve a reasonable bandwidth when sent over a network.

In this work, we characterize the latency components of edge video streaming with the goal of identifying latency bottlenecks. In edge applications, the processing is performed at the edge of the network close to data generation, rather than in the cloud, to meet the latency requirements. This work specifically investigates the latencies in the transmit and receive paths in the Linux networking stack, and the impact of encoding parameters on the latency.

ACKNOWLEDGEMENTS

I would like to express my sincere gratitude to my advisor Dr. Arun Ravindran for all his help in pursuing this research and developing this thesis. I would also like to thank the other members of my committee Dr. Tao Han and Dr. Hamed Tabkhi for their insightful comments and for taking the time to review this work.

I am forever thankful to my parents for their love and support throughout my life. I am also grateful to my whole family and friends for their love, encouragement, prayers, and support.

Finally, I would like to thank the Libyan Ministry of Higher Education for funding my studies.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1: INTRODUCTION

In recent years, there has been a huge demand in the use of streaming videos in various applications [1] [2]. These applications include video communication, video conferences, video games, surveillance, and the recent use of artificial intelligence, such as for real-time object recognition required for self driving vehicles [3].

Streaming video applications have three main components: a video encoder that compresses the video captured by a camera to reduce the transmission bandwidth requirements, a network for sending and receiving the encoded video, and a decoder that decompresses the received video. The decoded video is either displayed (for example, in video conferencing) or analyzed (for example, in surveillance).

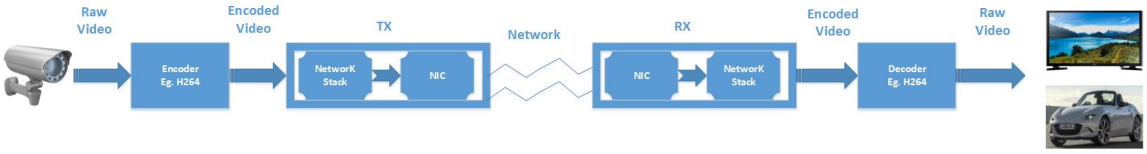

Figure 1.1: End-to-end Streaming Video Application.

Many of the streaming video applications are latency sensitive. The latency required depends highly on the application. For example, while 100 ms is sufficient for video conferences and video games [4], self driving vehicular requires latency of less than 10 ms [5]. Depending on the application, the latency figure could be the average latency or the tail latency such as the 99-th percentile latency.

In this thesis, we experimentally characterize latency components for video streaming with the goal of identifying latency bottlenecks. While prior work has characterized end-to-end latencies of cloud video streaming services [6], our work specifically focuses on the recently emerging edge video streaming applications (for example self

driving cars). In edge computing, the data computation occurs at the edge of the network close to data generation, rather than in the cloud, so as to meet latency requirements [7]. The video captured by a camera is encoded and transmitted to an edge micro-server over a local network, where it is decoded for further processing. The thesis investigates latencies in the transmit and receive paths of the Linux networking stack, and the impact of the encoding parameters on latency. Note that the transmit path is part of the camera system software, while the receive path is on the edge micro-server. Characterization of network latencies (from the transmit NIC to the receive NIC), while part of the overall end-to-end latencies, is outside the scope of the thesis.

1.1 Thesis Contributions

In summary, our main contributions in this work can be listed as follows:

- We presented the path followed by the packets inside the Linux networking stack.
- We conducted in-depth analysis of the latency in the networking stack and identified that the major part of the latency is in the boundary between user and kernel spaces.
- We conducted analysis to measure the latency caused by the encoding process, and analyzed various parameters that can be adjusted to reduce this latency.

1.2 Thesis Organization

In this chapter, we provided an overview of the edge video streaming applications and presented the research problem. In chapter 2, we present the tools and utilities that have been utilized in this work. In chapter 3, we discuss the flow of the packets in the networking stack and break down the latencies in the stack for both the transmission and receive. In chapter 4, we present the latency results for the encoding process and analyze how the various ffmpeg settings can be adjusted to reduce this latency. In chapter 5, the conclusion of the work is presented.

CHAPTER 2: TOOLS AND UTILITIES

In this chapter, we provide a brief description of the tools used in this thesis. Some of these tools were used for debugging and measurements, such as Ftrace, Strace, and SystemTap. Others, such as sysctl and ethtool, were used to examine and modify the settings in the kernel, driver, and NIC.

2.1 Ftrace

Ftrace is a tool used for kernel debugging. It utilizes the debugfs virtual file system, and the control files of the tool reside there. This tool was mainly used in this work to trace the path taken by the packets in the Linux networking stack. In order to be able to timestamp the different points in the stack, we needed to know the exact names of the functions. Since these names are not documented well and slightly change between the different Linux versions, we used Ftrace to find the functions called in the different layers in the stack. Ftrace by default probes all the functions in the kernel. Using Ftrace for tracing during high traffic would very likely result in hogging the system as there is extra work to be done whenever any function is called while the system is already highly utilized. For this reason, we used Ftrace during low traffic as the same functions will be called whether there is low or high traffic. However, it is possible to track only certain functions by using the tool's filtration mechanisms. The filtration is performed by writing the function names into a special file called set_ftrace_filter. We can examine the functions that currently being probed by viewing the $available\_filter\_functions$ file [8].

2.2 Strace

Strace is a system call tracer that is mainly used to observe and debug the behavior of an application by probing the interaction between the user and kernel spaces. By observing the arguments passed to the system calls and their return values, we can characterize the application's behavior and speculate what it demands the kernel to execute on its behalf. This tool can also measure the number of times each system call is issued and the duration this syscall spent in execution [9].

2.3 Tcpdump

Tcpdump is a command-line packet sniffing tool that uses the libpcap library. We used tcpdump to dump the packets into a file as we were accessing the server remotely, then we used Wireshark for post processing analysis. This tool provides filtration options, such as sniffing the packets that belong to a specific port number or IP address [10].

2.4 Wireshark

Wireshark is a packet sniffing tool that has a graphical interface and provides a visual analysis of the packet. For example, we can examine directly whether or not the packet is fragmented by examining the MF flag in the IP header. In addition, Wireshark has a filtering capability; for example, we can filter the packets by the IP address or/and the port number of either the source or destination. Furthermore, the coloring filtering tool makes it effortless to spot the erroneous packets [11].

2.5 SystemTap

SystemTap is a debugging and performance measurement tool that allows on-the-fly kernel programming without the need to reboot or recompile the kernel each time we add a new module. The scripts of SystemTap are written in a special scripting language, which is then compiled and loaded into the kernel as a module. The module is removed at the end of running the script. SystemTap can probe most of the

functions in the kernel code and inspect their arguments, local variables, and return values. In addition, we can add some extra code to be executed when a particular function is called or when it is returning. Since we are dealing with high traffic, some functions in the networking stack will be executed millions of times. So, we had to be careful with the code we write to be executed when a function is called. For example, if we try to print a value inside one of the frequently called functions, then the server will hang, and we will have to reboot it [12].

2.6 Sysctl

Sysctl can be used to examine and adjust the kernel parameters in the */proc/sys/* directory at runtime. The parameters changed by sysctl does not persist after reboot. To make the changes persistent, we have to add the values to */etc/sysctl.conf* file. The *init* process will execute this file at boot time to determine the values that should be assigned to the kernel parameters. An example of a parameter can be adjusted using sysctl is *dev_weight*, which is the maximum number of packets that NAPI can process for a particular interface. The parameter *dev_weight* resides in the following directory */proc/sys/net/core*, so it can be modified by: [13]

```
1 $sysctl -w net.core.dev_weight=<weight>
```

2.7 Ethtool

Ethtool is a utility to view and modify the settings of the NIC and the network device driver. For example, it can be used to view and change the speed, the duplex, and the offload features of the interface. The tool also provide some statistics and diagnostics for both the NIC and the driver and the parameters supported depends on both the NIC and its driver [14].

CHAPTER 3: THE NETWORKING STACK LATENCIES

We begin this chapter by presenting a brief overview of the networking protocols and codec standards used in this work. Next, we trace the flow of the packets in the networking stack and identify the key functions executed during the transmit and receive of packets. We then measure the latencies in the various parts of the networking stack for both the transmission and receive.

The experiments in this chapter are performed using Intel W2600CR and Intel SandyBridge machines. The two machines are connected directly to each other by an Ethernet cable. Each of the machines have 32 cores and 1 Gb/s igb Intel NIC. Both of the machines run Linux version 4.4.0. ffmpeg was used to encode and send the frames and $ffplay$ was used to receive and decode the frames but without displaying them.

3.1 Network and Video Encoding Protocols

In this work, The Real-Time Transport Protocol (RTP) is used on top of the User Datagram Protocol to transmit and receive the video frames. The H.264 codec standard is used for encoding and decoding the videos.

3.1.1 Real-Time Transport Protocol (RTP)

RTP provides end-to-end network transport functionalities suitable for real-time applications. RTP is not a transport layer protocol, so it is usually used with UDP or TCP. The RTP standard defines a pair of protocols: RTP and RTCP. RTP is used for transporting the data, while RTCP is used to periodically receive feedback about the quality of the transmission. RTP does not reserve bandwidth or guarantee quality of service. However, there are application layer control protocols, such as H.245, SIP,

and RTSP, that can be used to negotiate the capabilities of the sender and receiver and to control the running session. The Real-Time Streaming Protocol (RTSP) is commonly used with RTP [4] [15].

The RTP header contains the following: the sequence number, which is incremented by one for each packet sent in a session and could be used for detecting packet drops; the timestamp, which provides timing information; the payload type, which identifies the type of media codec of the payload; and the marker bit, which specifies the last packet in the frame [4].

3.1.2 User Datagram Protocol (UDP)

UDP offers a simple, unreliable datagram transport mechanism. The only error detection mechanism provided by UDP is checksumming which checks if the packet is erroneous. So, packets can be lost, duplicated, or arrived out of order in their path from source to destination. The mechanisms to deal with those situations should be provided by upper layers if needed. TCP on the other hand provides guaranteed transport service by retransmitting the lost and the erroneous packets [4].

UDP has been used as the transport layer protocol in this work because it is suitable for real-time applications. UDP is faster than TCP due to its minimal header, lack of acknowledgment, and lack of packet retransmission. Retransmission of packets is usually not useful in real-time applications [16]. Using UDP yields an improvement even in non real-time applications. For example, Facebook achieved 20% speedup in Memcached when UDP was used instead of TCP in handling some requests [17].

3.1.3 H.264 Video Codec Standard

Video coding is an integral part of many visual information based applications. There are two main parts in this process: The first part is the encoding, which compresses the video to reduce its size so it can be stored on disk or sent through network efficiently. The second is the decoding, which decompresses the encoded

video to be displayed or processed further.

The H.264 is also known as MPEG-4 Advanced Video Codec (AVC), and it is a product of the joint video team (JVT). Its design is based on the distinction between the video coding layer (VCL) and the network adaptation layer (NAL). The VCL deals with the compression of data, and comprises syntactical levels such as the block, macroblock, and slice level. the NAL organizes the compressed data in order to be stored or transmitted [4].

H.264/AVC standard has been chosen as the codec for this study because it is one of the most efficient video standards [18] [19] [20] [21]. It provides better coding efficiency, and gives lower bit rates than the previous standards such as MPEG-2 and H.263 [21]. H.264 is widely adopted in various multimedia devices, ranging from mobile phones to high-definition television [19]. This codec has also been used for low-latency applications [22].

3.2 Network Driver Initialization

The driver's initialization is performed either during the boot or when its module is being inserted into the kernel. There are several tasks that needs to be performed before any packets could be received. These include allocating the receive and transmit ring buffers, initializing NAPI, enabling the hardware interrupts, and registering an interrupt handler to be executed each time the designated irq is received.

3.2.1 The Ring Buffers

The structure of the transmit and receive descriptors, which hold the metadata of each buffer such as its address and length, is prescribed in the NIC manual. The driver allocates memory to store these buffers and descriptors in RAM, which are shared by both the NIC and the driver. For example, the NIC copies the received packet to the receive ring buffer using DMA and then updates the associated metadata.

3.3 The Packets' Path inside the Networking Stack

There are a few resources describe the path taken by the packets inside the networking stack [23] [24] [25] [26] [27]. Unfortunately, these sources either describe the flow for old versions of Linux (version 2.6 or older) or provide very short description that are not sufficient for our purpose of breaking down the latencies of the networking stack which require the knowledge of the exact names and order of the functions. For this reason, we used a function tracer called Ftrace to find the exact path that the packets take inside the stack. We provide an overview of the path taken by the packets in the networking stack which can be used as a general reference, and will help us in making decisions as to where to put the probe points which is our main goal. Since there are significant number of functions in the stack, we will not be discussing all of them. For example, in the IP layer, there are other functions that might be called such as the functions related to the NF Hook filter, NAT, and iptables; besides the fragmentation functions if we have large packets.

3.3.1 Identifying the Functions Executed In the Stack

The number of functions executed in the kernel is enormous! We couldn't filter for specific functions because we are using Ftrace to actually find the names of the functions. Tracing even for a few seconds gave millions of functions, and soon enough the output file of the tool would reach its maximum limit, and we could not even see the functions related to the networking stack. A few approaches were used to solve the problem: First, we monitored the functions for the duration needed to execute only one command. We chose the ping command, and we pinged the loopback as it is quite fast. This approach worked well, but it was somewhat limited for two reasons: first, the loopback does not access the NIC, so we couldn't see the functions related to interrupts for example. Second, the ping command uses ICMP over IP, and we are interested in the UDP protocol rather than ICMP. Using the ping command to ping

a direct connected computer solved the first problem. So now we were able to see all the functions related to the hardware interrupts and softirq. To reduce the context switches, we assigned the highest priority to the process that executes the networking protocol using the "nice" command. To reduce the unnecessary functions from the other cores in the server, we turned off all the core except one core. This reduced the number of functions unrelated to the networking stack significantly and enabled us to follow the path of the packets in the stack. The following bash script disable all the 32 cores in the server except core 0.

```
1 for i in ‘seq 1 31‘;
2         do
3                 echo 1 | sudo tee /sys/devices/system/cpu/cpu$i/online
4         done
```

3.3.2 Identifying the packets

The number of the packets received by a gigabit link is enormous and can be processed by multiple CPUs simultaneously. In order to make accurate measurements and measure the latency of each packet individually, we needed a way to identify the packets. We identified the packets using the identification field in the IP header which was extracted from the skbuff structure. However, since there is only 16 bits in this field, the id wraps around after 65,535 packets. We solved the wrapping around problem by storing the measured timestamps in an array indexed by the id field; then, the latencies were calculated every 1 ms, as the packets won't wrap around in this very short period.

3.3.3 The Receive Side

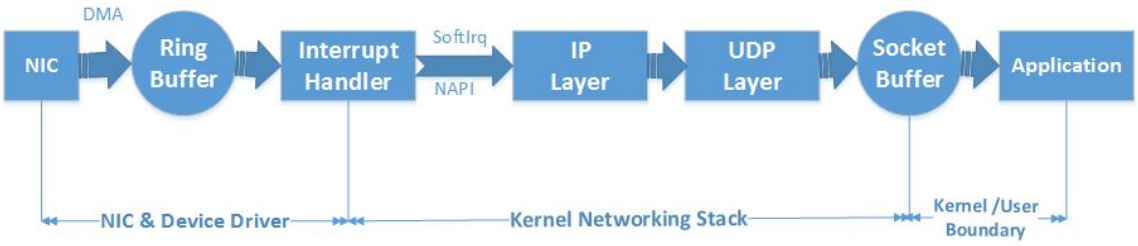

Figure 3.1: The Receive Side.

3.3.3.1 The Interrupt Handler

The snippet shown below is for the functions executed in the interrupt handler. The interrupt handler is very short and takes minimal time to execute. It raises the softirq flag where the heavy work is done. In addition, the hardware interrupt handler wakes up the NAPI softirq process. As long as we are not using the receive packet steering, the NAPI and softirq will be executed on the same CPU as the hardware interrupt. The number of interrupts issued is significantly less than the number of received packets. This interrupt coalescing improves the performance during high traffic. If we have an interrupt per packets for gigabit links, we might end up with a livelock where the processor is just busy servicing the interrupts and cannot process the packets. A higher coalescence setting will result in fewer interrupts generated, which in turn will result in higher throughput, lower CPU usage, and lower power consumption. This is mainly because it will reduce the number of context switches needed for interrupt handling. However, it will cause the latency to increase as the packets will wait longer before being handled [23]. The time taken to execute the interrupt handler is about 1 μs on average. The principle is to defer as much processing as possible to happen outside the hardware interrupt context. We can view statistics about the hardware interrupts and which CPUs handles which interrupt by accessing the following file: */proc/interrupts*.

Figure 3.2: The Interrupt Handler.

3.3.3.2 The Softirq

The softirq could start running right after the hardware interrupt is handled or in the context of the *ksoftirqd* daemon. There is one *ksoftirqd* per CPU, and they are registered during boot. Each softirq has a specific handler associated with it, the handler for the Ethernet receive is *net_rx_action*(). The *__raise_softirq_irqoff*() function which called during the interrupt handler marks the NET_RX_SOFTIRQ softirq as pending and wakes up the softirq thread on the current CPU to execute the associated handler. The status of all softirqs and how many times they have been executed on each CPU can be found in */proc/softirqs* [26].

3.3.3.3 *net_rx_action*()

This function starts the NAPI processing loop to handle the packets that have been DMAed to the ring buffer one at a time. This is done by browsing the *poll_list* of the devices that have something in their ingress queue, and invoking the associated *poll*() for each one until one of the following conditions is met:

• There are no more devices in the list, which means all the packets in all interfaces have been handled.

• *net_rx_action*() has run for too long and therefore it is supposed to release the CPU. This occurs when the number of packets already dequeued and processed has reached a given upper bound limit called the budget [23]. The default budget is 300 packets, but it can be changed using:

```
1 $ sysctl net.core.netdev_budget=<desired number of packets>
```

3.3.3.4 The New API (NAPI)

NAPI combines interrupts with polling to give higher performance under high traffic by reducing significantly the load on the CPU. The load on the CPU is reduced by minimizing the number of interrupts, as each interrupt takes time to be handled and requires two context switches [23]. NAPI achieves fairness in handling the packets of different interfaces by handling only specific number of packets from each interface and then iterates through the rest of the interfaces in a round robin fashion. This ensures that the interfaces with low traffic can experience acceptable latencies. The default value of the limit of each interface is 64 and it is called *dev_weigh*, and can be adjusted using:

```
1 $ sysctl -w net.core.dev_weight=<desired interface limit>
```

napi_schedule() is called to wake up the NAPI if it is not already active, and it is called as part of the hardware interrupt handler. However, the NAPI processing loop executes in a softirq context. During the driver initialization, the driver registers a *poll*() function for each interface and assigns a weight for it. The *poll*() function returns the number of packets that has been dequeued, which could be equal to the weight or less. The *net_rx_action*() will subtract this from the budget.

3.3.3.5 The Socket Buffer (skbuff or skb)

The socket buffer is a data structure used by various layers and protocols in the Linux networking stack. The stack layers uses the same structure to avoid unnecessary copying. For example in the transmission, the socket allocates an skbuff structure and copies the payload data to it, then the transport layer adds its header and so does the network layer, and finally the data link layer adds the MAC header. The kernel maintains all skbuff structures in a doubly linked list [23]. The skbuff contains a lot of information about the packet such as its length, and pointers to its payload and headers. $\_\_napi\_alloc\_skb()$ is the function responsible for creating and building the skbuff structure in the receive side. It calls $\_\_build\_skb()$ which takes an skbuff structure from the cache by calling the function $kmem\_cache\_alloc()$. The snippet from Ftrace below shows the main functions discussed above.

```
0)               |                                  __do_softirq() {
0)               |                                    net_rx_action() {
0)               |                                      igb_poll [igb]() {
0)   0.074 us    |                                        igb_update_dca [igb]();
0)               |                                        igb_clean_rx_irq [igb]() {
0)               |                                          __napi_alloc_skb() {
0)   0.066 us    |                                            __alloc_page_frag();
0)               |                                            __build_skb() {
0)   0.104 us    |                                              kmem_cache_alloc();
0)   0.608 us    |                                            }
0)   1.585 us    |                                          }__
```

Figure 3.3: The NAPI Processing Loop.

3.3.3.6 The Network and Transport Layers

The protocol is identified by a call to $eth\_type\_trans()$. Then, the packet is delivered to the upper protocol layers using $\_\_netif\_receive\_skb\_core()$. In case of IP protocol, it will deliver the packet to $ip\_rcv()$. $ip\_rcv()$ is the first function in the network layer and mainly checks for errors such as the header length and the checksumming. It calls $ip\_rcv\_finish()$ which checks whether this device is the destination. If so, it calls $ip\_local\_deliver()$. If this device is not the final destination, then $ip\_forward()$ will be called. The packet is subsequently passed to

the UDP layer, starting with *udp_rcv*(). The code for this function is just one line that calls directly *__udp4_lib_rcv*(). *__udp4_lib_rcv*() will validate the packet and lookup the socket of which the packet belongs using *__udp4_lib_lookup*(). The packet is then enqueued to the socket buffer using *sock_queue_rcv_skb*() which checks for the security permission then pushes the packet into the socket buffer. Before adding the packet to the socket queue, a check is made to see if there are any syscalls issued by the application to receive from this socket. If there is a blocked receive system call, then the packet is delivered to the user space. Otherwise, the packet is enqueued into the socket buffer [25] [28].

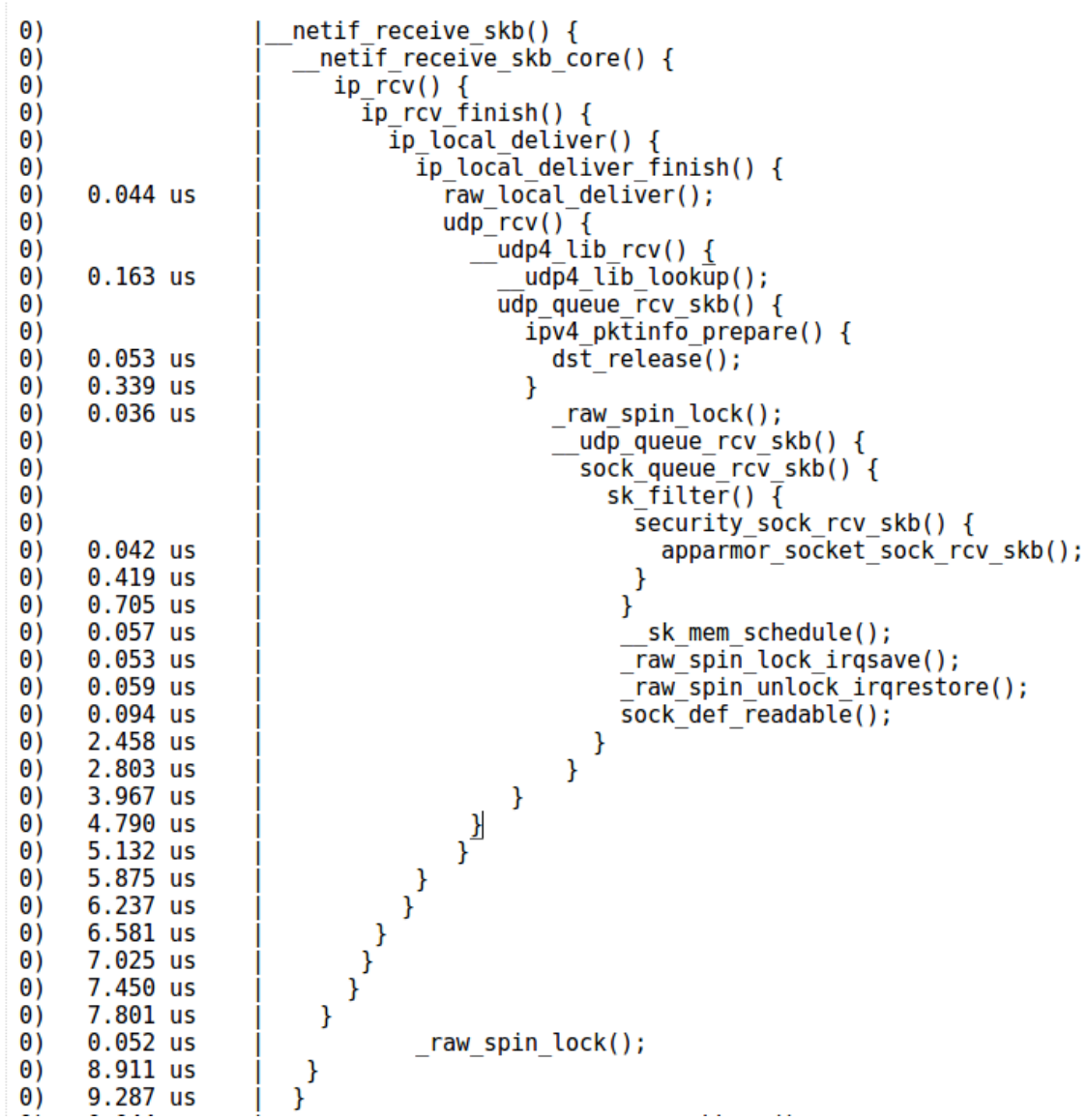

Figure 3.4: The Network and Transport Layers.

3.3.3.7 Harvesting the Packets

The packets will remain in the socket's queue until the user issues a system call to harvest the packets using the Linux APIs such as *read*(), *recv*(), *recvfrom*(), *recvmsg*(), or *recvmmsg*(). Usually *poll*(), *select*(), or *epoll*() are used first to determine if one of the sockets in the group has data in its queue. *epoll*() has a performance advantage over *poll*() when there are large number of sockets. All of the receive syscalls will end up in *sock_recvmsg*() which checks that the caller has the right

permission by calling *security_socket_recvmsg*(), and then calls the receive function associated with the protocol which is *udp_recvmsg*() for UDP. *udp_recvmsg*() will deliver the packet to the user and then call *__kfree_skb*() to free the skbuff structure [26]. If the user space harvests the packets faster than they arrive, then the issued system call will either block or return a negative value depending on the attributes passed to it. Notice that generally if the system call successfully harvests a packet, it will return the number of bytes that have been read from the socket. Issuing a system call and the associated context switches from user to kernel and to user again can be somewhat expensive. Consequently, *recvmmsg*() was introduced to harvest multiple packets using one system call [25].

```
0)               |  sock_poll() {
0)   0.162 us    |    udp_poll();
0)   0.971 us    |  }
0)               |  SyS_recvfrom() {
0)   0.143 us    |    sockfd_lookup_light();
0)               |    sock_recvmsg() {
0)               |      security_socket_recvmsg() {
0)               |        apparmor_socket_recvmsg() {
0)   0.272 us    |          aa_sock_msg_perm();
0)   1.159 us    |        }
0)   2.069 us    |      }
0)               |      inet_recvmsg() {
0)               |        udp_recvmsg() {
0)   0.220 us    |          __skb_recv_datagram();
0)   0.484 us    |          skb_copy_datagram_iter();
0)               |          skb_free_datagram_locked() {
0)   0.119 us    |            lock_sock_fast();
0)   0.119 us    |            sock_rfree();
0)               |            __kfree_skb() {
0)               |              skb_release_all() {
0)   0.114 us    |                skb_release_head_state();
0)   0.191 us    |                skb_release_data();
0)   1.928 us    |              }
0)   0.198 us    |              kfree_skbmem();
0)   3.789 us    |            }
0)   6.937 us    |          }
0) + 10.037 us   |        }
0) + 10.877 us   |      }
0) + 14.510 us   |    }
0) + 16.553 us   |  }
```

Figure 3.5: Harvesting the Packets Using $recvfrom()$.

3.3.4 The Transmission Side

Figure 3.6: The Transmission Side.

The transmission starts with a Linux API such as $send()$, $sendto()$, or $sendmsg()$. These functions issue a system call to transit from the user to the kernel space. After looking up the socket, $sock\_sendmsg()$ is called to check the permissions. It then calls the appropriate functions depending on the protocol; for example, $inet\_sendmsg()$

and then $udp\_sendmsg()$ for UDP. $udp\_sendmsg()$ will allocate an skbuff and then call $ip\_route\_output\_flow()$ to find the route that the packet will take. In addition, the UDP layer and the IP layer will append their headers. Afterwards, $dev\_queue\_xmit()$ enqueues the packet into the queuing discipline (Qdisk). There are several scheduling policies that can be used with the Qdisk. The default policy is called $pfifo\_fast$ which is a FIFO with three priorities. The packet's priority depends on the TOS field in the IP header [29]. Devices that don't have a Qdisk such as the loopback directly call $dev\_hard\_start\_xmit()$. $dev\_hard\_start\_xmit()$ dequeues the packet from the Qdisk and puts it in the ring buffer and updates the relevant descriptors. Now the NIC can copy the packet to its own queue using DMA. Both the scheduling policy and the length of the Qdisk can be adjusted. The length of the Qdisk is called $txqueuelen$ and its default value is 1000 packets [29], but can be changed using:

```
1 $ ifconfig eth0 txqueuelen <desired length>
```

```
0)               |  SyS_sendto() {
0)               |    sockfd_lookup_light() {
0)               |      __fdget() {
0)   0.032 us    |        __fget_light();
0)   0.290 us    |      }
0)   0.537 us    |    }
0)   0.086 us    |    move_addr_to_kernel.part.14();
0)               |    sock_sendmsg() {
0)               |      security_socket_sendmsg() {
0)               |        apparmor_socket_sendmsg() {
0)               |          aa_sock_msg_perm() {
0)               |            aa_sk_perm() {
0)   0.058 us    |              aa_label_sk_perm();
0)   0.427 us    |            }
0)   0.683 us    |          }
0)   0.937 us    |        }
0)   1.246 us    |      }
0)               |      inet_sendmsg() {
0)               |        udp_sendmsg() {
0)   0.063 us    |          security_sk_classify_flow();
0)               |          ip_route_output_flow() {
0)               |            __ip_route_output_key_hash() {
0)   1.705 us    |              fib_table_lookup();
0)   0.039 us    |              find_exception();
0)   3.144 us    |            }
0)               |            xfrm_lookup_route() {
0)   0.106 us    |              xfrm_lookup();
0)   0.435 us    |            }
0)   4.103 us    |          }
0)               |          ip_make_skb() {
0)               |            ip_setup_cork() {
0)   0.121 us    |              ipv4_mtu();
0)   0.456 us    |            }
0)               |            __ip_append_data.isra.42() {
0)               |              sock_alloc_send_skb() {
0)               |                sock_alloc_send_pskb() {
0)               |                  alloc_skb_with_frags() {
0)               |                    __alloc_skb() {
0)               |                      kmem_cache_alloc_node() {
0)   0.032 us    |                        _cond_resched();
0)   0.331 us    |                      }
```

Figure 3.7: The Transmission Side, Part1.

```
0)               |          udp_send_skb() {
0)   0.044 us    |            udp4_hwcsum();
0)               |            ip_send_skb() {
0)               |              ip_local_out() {
0)               |                __ip_local_out() {
0)   0.028 us    |                  ip_send_check();
0)   0.401 us    |                }
0)               |                ip_output() {
0)               |                  ip_finish_output() {
0)   0.028 us    |                    ipv4_mtu();
0)               |                    ip_finish_output2() {
0)   0.031 us    |                      skb_push();
0)               |                      dev_queue_xmit() {
0)               |                        __dev_queue_xmit() {
0)   0.169 us    |                          netdev_pick_tx();
0)               |                          validate_xmit_skb.isra.97.part.98() {
0)               |                            netif_skb_features() {
0)   0.034 us    |                              skb_network_protocol();
0)   0.502 us    |                            }
0)   0.856 us    |                          }
0)               |                          dev_hard_start_xmit() {
```

Figure 3.8: The Transmission Side, Part2.

3.3.5 Dropping Packets

During high traffic, there is a possibility of packet drops. The first thing to do to handle this problem is to find out where the packet drops occur. Fortunately, all the packets dropped in the networking stack call the same function to drop the packets; this function is $kfree\_skb()$[23]. By probing this function and backtrace the location from where it is called, we can determine which function in the stack dropped the packet. For example, if the user's application is slow in handling the packets and does not issue system calls to harvest the packets frequently enough, then the socket queue will get full and the packets will be dropped. This is occurred when we used ffmpeg because decoding the video frames were taking significant time. To solve this problem, we first identified the location where the packets being dropped using SystemTap where we found that $udp\_queue\_rcv\_skb()$ was the function that is dropping the packets. Then we looked at the kernel's source code to see where in that function the packets might be dropped. By examining the several places where the packets are dropped in that function, we found that the reason is because $sk\_rcvqueues\_full()$ was returning a boolean true value. Examining the source

code of the latter function confirmed that the socket receive queue is full. The limit variable is `$sk->sk_rcvbuf` as we can see in the source code below. By tracing this variable and examining its value using SystemTap, we confirmed that this is the value of the socket buffer which can also be read from */proc/sys/net/core/rmem_default*. Since *sk_rcvqueues_full*() is an inline function, it cannot be probed directly, so we probed the function that calls it instead which is *udp_queue_rcv_skb*().

```
if (sk_rcvqueues_full(sk, sk->sk_rcvbuf)) {
    UDP_INC_STATS_BH(sock_net(sk), UDP_MIB_RCVBUFERRORS,
                     is_udplite);
    goto drop;
}
```

Figure 3.9: A snippet from *udp_queue_rcv_skb*().

```
static inline bool sk_rcvqueues_full(const struct sock *sk, unsigned int limit)
{
    unsigned int qsize = sk->sk_backlog.len + atomic_read(&sk->sk_rmem_alloc);

    return qsize > limit;
}
```

Figure 3.10: The Source Code of *sk_rcvqueues_full*()

Increasing the socket buffer supposedly can be done from the command line using sysctl. However, this did not work for the Linux 4.4.0 used in this work although reading the buffer value from the *proc* file system at */proc/sys/net/core/rmem_default* assured it has changed. However, When we read the `$sk->sk_rcvbuf` which holds the socket buffer size using SystemTap, we found that strangely enough the value is fluctuating between the old value of the socket buffer and the new value assigned by sysctl. To fix this, we modified the value permanently by writing it into */etc/sysctl.conf* file and then executed $*sysctl* − *p*/*etc*/*sysctl.conf*. After this, the socket buffer value were set at the boot and didn't fluctuate. Increasing the socket buffer might not work if the application tries to adjust it using *setsockopt*(). For example, we noticed that the socket buffer size changes to a smaller value only when we ran the ffmpeg

application. Further investigation of the ffmpeg source code has shown that ffmpeg sets the buffer size using *setsockopt*() with the SO_RCVBUF option. Commenting this function in the application's source code and recompiling it fixed the problem.

3.3.6 Breaking down the Latencies of the Receive Side

We measured the average latency alongside with the 99th percentile latency (tail latency). The tail latency is quite important in applications where the single request is sent to multiple servers, and the decision cannot be made until all the request have been serviced. In such cases, the average time is not an accurate measure because the actual response time depends on the slowest server [30]. Table 3.1 shows the latencies in the main parts of the stack. The first entry is for the interrupt handler which takes a quite short time, as it mainly acknowledges the interrupt and raises the softirq flag where the heavy processing will be done. There are many types of interrupts and all of them are using the same initial function *handle_irq*(). To be able to filter and measure only the interrupts of the ethernet interface under test, we filtered the interrupts by the irq number. Since our NIC uses multiple queues, we included all the irqs from 56 to 64 where each one of them is associated with one of the NIC queues. The second entry in Table 3.1 is the latency in the IP and the UDP layers. The latency in these two layers is only a few microseconds. The reason behind this is that the UDP layer, which has a header of only 8 bytes, is quite thin and does not do much processing. TCP, unlike UDP, is doing some complicated work like error recovery, flow control, and packet acknowledgement. The cost of the UDP efficiency is that the application has to implement its own flow control and error recovery if required. The latency of enqueuing the packet in the socket is quite small as well. This latency is simply the duration of the function *sock_queue_rcv_skb*().

Table 3.1: Breaking down the Latency in the Networking Stack.

The location	Average latency (μs)	Tail latency (μs)
Interrupt handler	1.23	2.31
IP and UDP layers	3.32	11.24
Socket enqueue	2.91	10.21
Socket enqueue to user space	5564.23	23462.78
$sys\_recvmsg()$	27.87	160.56

Next, we measured the latency right after enqueuing the packet into the socket until it is delivered to the user space. This latency was quite huge compared to the other parts, so we investigated further to characterize why the latency is very high. First, we measured the time that the system call takes to execute since the boundary between user space and kernel space has been blamed of causing high latency [31] [32]. There were even quite a few suggestions to improve this latency by making a zero copy networking stack [32] [33]. Although the latency of the receive system call was a few times larger than the latency in the other parts of the stack, it represented only a fraction of the huge latency observed in the earlier measurement. Since there is no processing done between enqueuing the packet into the socket and started harvesting it using the syscall, the only explanation to this latency is that the application is not issuing system calls fast enough. This is quite logical as the ffmpeg does spends significant time in decoding the frames.

To prove the previous assumption, we used a trivial application that simply does not do any noticeable processing [34]. It simply puts the received packets in a buffer and then discards them. Measuring the latency from the moment the packet enqueued into the socket until it is delivered to the user space for this application still gave a large value, but not as huge as the ffmpeg measurement as can be seen in Table 3.2. Observing the functions that caused this high latency in the trivial application case,

we found that the function *schedule*() is responsible for this latency. This is because the syscall is blocking when there is no packets in the socket queue, so the scheduler simply schedules other work. So, this is rather a false latency because there are no packets actually waiting during this time.

Table 3.2: Comparison Between ffmpeg and a Trivial Application.

The location	ffmpeg		Trivial app.	
	Avg. Lat. (μs)	Tail Lat. (μs)	Avg. Lat. (μs)	Tail Lat. (μs)
Socket enqueue to user space	5564.23	23462.78	323.46	897.32
sys_recvmsg()	27.87	160.56	56.32	449.1

3.3.7 Jumbo frames

The jumbo packets are packets larger than the default size of 1500 bytes; the most common size of jumbo frames is 9000 bytes. The main advantage of jumbo frames is reducing the overhead [23]. For example, if we assumed a packet with 46 bytes of headers (18 bytes in the data link layer, 20 bytes in the IP layer, and 8 bytes in the UDP layer). With 1472 payload, which is the maximum, as the 1500 limit must include both the network and transport headers, the overhead is 3%. However, with 8972 as payload, the overhead is only 0.5%. The main drawback of jumbo frames is that not all network devices fully support it yet. However, most of the Gigabit routers, switches, and NICs support jumbo frames although the default is still mostly the 1500 bytes. The default value of the Maximum Transmission Unit (MTU) is 1500 bytes excluding the ethernet header. This setting must be increased in order to use jumbo frames which can be done using the ifconfig utility. Comparing the receive side latency of the IP and UDP layers of using non-jumbo frames (Table 3.1) and using jumbo frames (Table 3.3) shows that the time required to handle a jumbo packet is quite larger. This is expected as the jumbo packet is six times larger. In

Table 3.3, we observe the effect of sending large packets while maintaining the default MTU of 1500. In this case, we need more work to reassemble the fragmented packets which resulted in higher latency. The duration of the receive system calls however is almost the same whether the MTU is 1500 or 9000. This is because fragmenting and reassembling the packets occur in the IP layer, so for the layers above that, the packet is already reassembled and there is no extra work to be done.

Table 3.3: The Latency Distributions When Using Jumbo Frames.

The location	MTU 1500		MTU 9000	
	Avg. Lat. (μs)	Tail Lat. (μs)	Avg. Lat. (μs)	Tail Lat. (μs)
IP and UDP Layer	63.21	189.67	15.35	59.24
Socket Enqueue	8.28	39.78	11.19	41.67
sys_recvmsg()	52.34	271.58	59.26	361.34

3.3.8 The Transmission Side Latency

The combined latency from the beginning of execution of the *sendto*() syscall in the kernel (*SyS_sendto*()) until the function that delivers the packet to the NIC (*dev_hard_start_xmit*()) is about 6.7 μs on average, while its tail latency is approximately 11 μs.

Table 3.4: The Transmission Side Latency.

Location	Avg. Latency (μs)	Tail Latency (μs)
SyS_sendto() to *udp_send_skb*()	2.71	4.025
udp_send_skb() to *dev_hard_start_xmit*()	3.92	6.95

3.3.9 Packet Fragmentation

We have seen in Table 3.3 that fragmenting a packet increases the latency in the IP layer. The latency when the MTU size was kept at 1500 was four times larger than

the latency when MTU was 9000. One approach to minimize this latency without changing the MTU is by offloading the fragmentation to the NIC. The modern NICs can handle the fragmentation and the reassembly of the packets of the commonly used protocols such as UDP and TCP. Performing these operations in the hardware is much faster than in the stack.

3.3.10 The Offloading Features

The NICs support several offload features where the NICs perform some operations in their hardware rather than performing them in the stack by the operating system. One example mentioned above is the fragmentation offload. Other features include the checksumming, generic segmentation offload, and large receive offload. These features helps in reducing the latency as the example of the fragmentation above shows. The support for these features varies widely among NICs and driver versions. However, the ethtool can be used to list the supported features using the `--show-features` attribute.

CHAPTER 4: The ENCODING LATENCIES

In this chapter, we analyze the impact of encoding parameters on the latency. The encoding process is computationally intensive, so in order to achieve real-time encoding, hardware implementation is often required [35] [36]. However, here we will use the ffmpeg implementation of the H.264 codec standard [37] to measure the encoding latencies in software. We tuned several parameters to reduce the latency, and observed the effect of these adjustments on the average frame latency and video quality. We changed one parameter at a time while keeping the default values of the other parameters.

4.1 Video Compression

Video compression involves reducing the video size by using fewer bits to represent the same video content. This is achieved by finding similarities within the frame itself, and similarities with neighbor frames. An uncompressed video has a constant bitrate based on pixel representation, image resolution, and frame rate. The H.264 encoder compresses the fast moving scenes more than the slow scenes, because humans are not able to distinguish all the details in the fast moving scene, while they might inspect the image of a slow scene more thoroughly [38].

4.1.1 Frame Types

Video compression utilizes different frames types in order to make the compression more efficient. There are mainly three types of frames: (1) I-frames which act as a reference, and they are usually the biggest in size since they do not use any other frame as a reference. (2) P-frames which can use the previous frames as a reference, so usually they are more compressible than I-frames. (3) B-frames can use both

the previous and latter frames as references, so they tend to be much lower in size compared to the other two types.

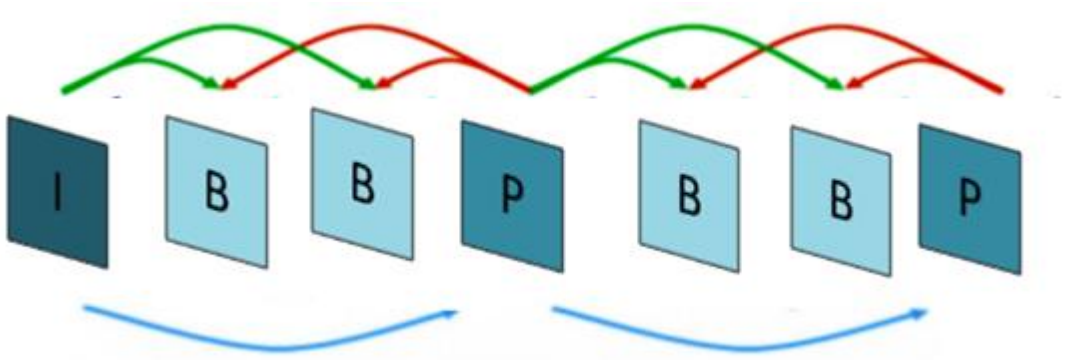

Figure 4.1: Typical Frame Sequence.

B-frames have to be encoded out of order since the latter P-frame has to be encoded first. This leads to extra latency. In latency-sensitive applications, B-frames are mostly avoided, and only I and P frames are used as shown in Fig 4.2

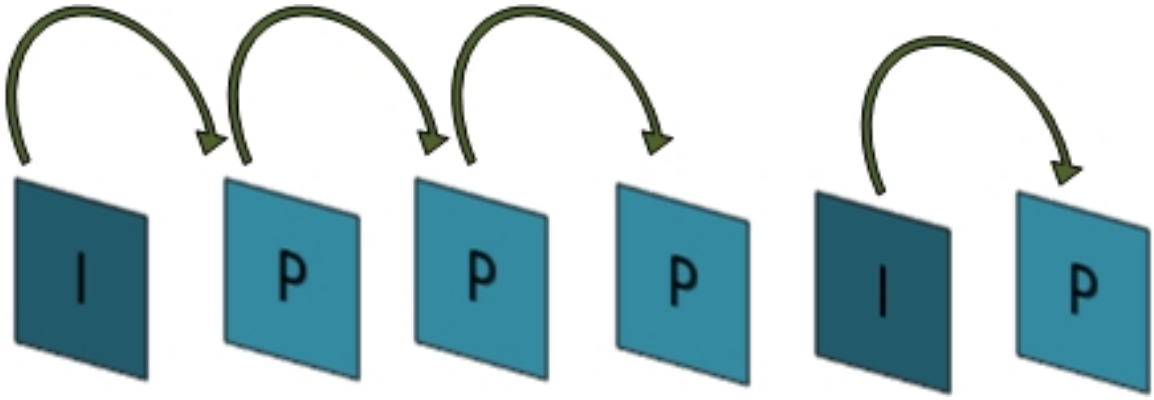

Figure 4.2: A Frame Sequence Without B-frames.

4.1.2 Video Quality

The video quality is quite important and can be measured in two different ways:

4.1.2.1 The Subjective Quality

Subjective quality is measured by the perception of the user. The problem with subjective quality is that it is quite subjective as it varies from one individual to

another, and also depends on the environment such as the amount of light in the room. Moreover, the human perception of the quality highly depends on the task, for example: watching a video content, video conference, or identifying a person in surveillance application [39].

4.1.2.2 The Objective Quality

The objective quality is calculated quantitatively such as by comparing the original video to the output video and measuring the noise ratio. There are two main methods used: the first is using pure mathematical modeling to measure the error as the difference between the original pixel and the processed one such as PSNR. The second method uses characteristics similar to the Human Visual System (HVS) such as SSIM [40].

• Peak Signal to Noise Ratio (PSNR) : In the context of video and image compression, the signal is the original image/video, while the noise is the error introduced by compression. Normally, higher PSNR means higher quality. The ideal value of PSNR is 100 dB, However, it ranges in practice between 30 and 50 dB [40]. The drawback of PSNR is that it looks only at individual frames, so it cannot take into account the perceptual effects such as motion.

• Structural Similarity (SSIM) : The SSIM is used to measure the similarity between two images. It is considered to be correlated with the human visual system.

In our study here, we will rely on the objective quality by using two metrics PSNR and SSIM to measure the quality. Measuring PSNR and SSIM using ffmpeg will increase the latency of the video slightly. To exclude this extra latency, all the latencies measured in this work are measured without including the objective quality metrics measurements. The quality metrics are measured separately.

4.2 Latency Measurements

4.2.1 Measurement Methodology

The sample video used in this work is a raw format 4k video that depicts nature scenes. The total number of frames is 179 and the frame rate is 30 frames per second. To measure the encoding latency, we need a way to measure the execution time of the ffmpeg application. First, we used strace to list all the system calls that are issued by this particular program. From the list of syscalls, we found that, *execve*() is the first system call to be called, and *sys_exit_group*() is the last one. Using strace, we used -ttt and -r flags to get the timestamping which provides microsecond granularity. However, we noticed a big overhead when measuring using strace. So we used SystemTap to make the measurements instead. The execution time measured by strace was more than six times the execution time measured by SystemTap. Clearly, the overhead of strace is unacceptable and most likely occurred because strace traces all syscalls even if in the command line arguments we specified only the two system calls that we are interested in probing. On the other hand, SystemTap traces only the functions that are probed in its script. All the measurements in this chapter are performed using SystemTap.

Since we measure the entry and exit of the program, the measurements will include context switches. To reduce this, we gave the application the highest possible priority in Linux which is -20 using "nice". This would limit the number of context switches, but it will not eliminate them. The average latency per frame is calculated by dividing the execution time that the ffmpeg takes to encode the whole video by the total number of frames in the video. Similarly, the average frame size is calculated by dividing the output file size by the total number of frames. The frame latency measured not only includes the encoding process, but also includes packaging the frames and passing them to the networking stack in the form of RTP packets.

4.2.2 The Encoding Latency

The main purpose of encoding is to compress the video to smaller size either to store it on the disk or to send it through a network. The important factors in encoding are: frame latency, frame size, and quality. The encoding process involves making trade offs between these factors. To show that the encoding latency is significant, we used ffmpeg to send a video. Initially, we did not change the codec ($-vcodeccopy$), and then we encoded the raw format video to H.264 standard. As shown in Table 4.1, The encoded operation took four times more execution time than the non-encoded one. However, the encoded video is 30 times smaller than the non-encoded one.

Table 4.1: The Encoding Latency.

	Avg Frame Lat.	Avg. Frame Size	File Size	Execution Time
Non-encoded	18.97 ms	2258.25 KB	404272 KB	3397 ms
H.264 Encoding	68.71 ms	71.36 KB	12775 KB	12300 ms

4.2.3 Constant Rate Factor (CRF)

This attribute lets us define the quality of the overall video. The range is from lossless (0) to the lowest quality (51). The default value (23) is near the middle [41]. The drawback of the CRF is that we do not know the average frame size or the output file size, which are very important to determine the bandwidth and the bitrate. CRF will try to maintain constant perceived quality throughout the video. This does not mean encoding all the frames with the same Quantization Parameter (QP), because this will result in higher quality for the fast motion scenes than the slow motion ones [41]. CRF instead will compress frames by different amounts depending on the type of motion in the scene. To set a minimum acceptable level of quality, $-crf\_max$ can be used. As shown in Table 4.2, we see that increasing the quality results in larger frame size and slower encoding. Choosing a lossless encoding resulted in a similar

average frame size as the original raw file, which is more than 30 times the resulted frame size from the default quality, it also resulted in higher latency.

Table 4.2: The Constant Rate Factor Settings.

CRF Value	Avg. Frame Latency (ms)	Avg. Frame Size (KB)	PSNR	SSIM
0 (lossless)	91.07	2259.89		
1	121.39	2166.67	60.37	30.88
20	79.43	138.43	45.87	15.30
23 (Default)	69.18	71.36	44.31	14.44
27	55.28	26.54	42.81	13.55
51 (low quality)	40.89	0.988	31.05	7.99

4.2.4 The Preset Setting

A preset is a setting that will provide a certain encoding speed to compression ratio. A slower preset will provide better compression [37]. There are several presets available for use with H.264 standard in ffmpeg. However, we will compare the extreme two ($ultrafast$ and $veryslow$). The $ultrafast$ is faster than the $veryslow$ setting by almost eight times, and it is faster than the default preset by two times. The cost is that the average frame size of the $ultrafast$ is 3.5 times larger than the $veryslow$, and 2.7 times larger than the default preset, which results in higher bitrate. The $ultrafast$ setting also counter intuitively results in higher quality video. In the $ultrafast$ preset, the speed is achieved by not using B-frames at all. B-frames are slower than P-frames, because they require out of order processing, although they result in very efficient video size. However, B-frames use significant approximations which degrades the video quality.

Table 4.3: The Preset Settings in H.264.

Preset	Avg. Frame Lat. (ms)	Avg. Frame Size (KB)	PSNR	SSIM
$Ultrafast$	34.11	194.54	45.32	14.79
$Medium$ (default)	69.04	71.38	44.34	14.43
$Veryslow$	256.00	54.97	44.04	14.31

4.2.5 The Tune Setting

There are several *tune* settings that ffmpeg uses with the H.264 standard such as *zerolatency*, $film$, and *animation*. We will focus on the setting related to latency which is *zerolatency*. The *zerolatency* setting is surprisingly slower than the default tuning setting. It also results in higher frame size though with slightly higher quality. Although *zerolatency* disables the B-frames like the $ultrafast$ preset does, it still gives higher latency even compared to the default. We do not observe any benefit of applying this setting.

Table 4.4: The Tune Settings in H.264.

Tune	Avg. Frame Latency (ms)	Avg. Frame Size (KB)	PSNR	SSIM
zerolatency	106.48	115.83	45.29	14.93
default	69.04	71.38	44.34	14.43

4.2.6 Threads Settings

Utilizing the multi cores in the machine by using multithreading yields significant improvement in speeding up the encoding process. The average frame size and the quality, as expected, are not affected by this setting. The main improvement observed when utilizing the 32 cores in the machine is reducing the latency by more than ten times compared to using only a single core. However, the default setting of H.264 in ffmpeg chooses the optimum number of threads depending on the machine

architecture, so there is no need to adjust this parameter.

Table 4.5: The Effect of Multithreading.

Threads	Frame Latency (ms)	Avg. Frame Size (KB)	PSNR
1 thread	749.31	71.39	44.34
16 threads	97.85	71.34	44.34
32 threads	72.27	71.30	44.35
default	68.29	71.36	44.34

4.2.7 Group of Pictures

This attribute determines the maximum distance between I-frames in the video [37]. A seen in Table 4.6, when choosing a small value of GoP such as 3, the latency is reduced by 1.5 times and the quality improved slightly, while the average frame size more than doubled. This is because I-frames are more accurate than B and P frames as an I-frame results from direct compression of the frame rather than comparing it with the neighbor frames. For the same reason, the frame size is larger. The latency reduction is because unlike P and B frames, I frames does not require any comparison with neighbor frames. Choosing smaller values of GoP is important when streaming a video to several clients, as it reduces the startup latency. In addition, smaller values of GoP will improve the seeking and fast forwarding in the player. The default value is 250 frames, which result in far distanced I-frames. For example, for 30 fps frame rate, we will have an I-frame only every 8.3 seconds [42].

Table 4.6: The Distance Between I-frames.

GoP	Avg. Frame Latency (ms)	Avg. Frame Size (KB)	PSNR	SSIM
3	45.17	155.17	45.09	15.12
15	60.10	91.32	44.64	14.65
30	64.05	81.89	44.54	14.55
250 (default)	68.71	71.36	44.34	14.43

4.2.8 Psychovisual Optimizations

This setting improves the subjective quality by trying to approximate the human visual system's perception [43]. As seen In Table 4.7, turning on this optimization slows down the encoding process and results in a bigger frame size. Although the objective quality metrics used did not show any improvements in quality, this is expected since this settings tries to improve the subjective quality. This setting should be turned on if the video is intended to be watched by humans, and should be turned off if the video will be supplied to a machine.

Table 4.7: Psychovisual Optimizations.

Psycho-Visual	Avg. Frame Latency (ms)	Avg. Frame Size (KB)	PSNR	SSIM
ON	68.28	71.36	44.34	14.43
OFF	61.49	55.88	44.43	14.60

4.2.9 The Resolution

The resolution of the raw input video is 4k (3840x2160), and the output will have the same resolution unless otherwise specified. Using lower resolutions such as 1080p and 720p resulted, as expected, in lower frame size and lower latency. The 720p resolution resulted in reducing the latency by half, and reducing the average frame size by thirteen times.

Table 4.8: The Resolution Settings.

Resolution	Avg. Frame Latency (ms)	Avg. Frame Size (KB)
720p (1280x720)	35.73	5.11
1080p (1920x1280)	39.34	15.41
4k (3840x2160))	69.04	71.36

4.2.10 Constant Bitrate (CBR)

There is no direct setting that can cause the bit rate to be constant. However, we can have the same effect by setting the minimum, average, and maximum bitrates to the same value [37]. This setting is useful only when we have a fixed bandwidth,and seek to utilize all of it. This setting also requires setting the VBV buffer. The CBR is not a good choice for storage as it will not allocate enough data for complex scenes while wasting data on the simple ones. As seen in Table 4.9, choosing a low bitrate speeds up the encoding process and gives smaller frame sizes but with largely degraded quality.

Table 4.9: Constant Bitrate Measurements.

Bit rate	Avg. Frame Latency (ms)	Avg. Frame Size (KB)	PSNR	SSIM
1Mb/s	45.31	3.93	34.92	8.99
10Mb/s	60.40	39.36	42.51	13.37
17Mb/s	67.03	66.92	43.50	13.95

4.2.11 Coders

The default coder in H.264 is Context-based Adaptive Binary Arithmetic Coding (CABAC). The other coder is Context-Adaptive Variable-Length Coding (CALVC) [44]. Both coders gives similar quality. However, CABAC, which is the default, gives better compression in terms of smaller frame size, but this comes at the cost of lower

speed. The latency of CABAC is slightly higher than CALVC as seen in the Table 4.10.

Table 4.10: Coder Types in H.264 .

Coder	Avg. Frame Latency (ms)	Avg. Frame Size (KB)	PSNR	SSIM
CABAC	69.04	71.38	44.34	14.43
CALVC	65.41	77.81	44.28	14.42

4.2.12 Motion-Estimation Search Pattern ($-motion-est$)

The encoding process seeks to find similar patterns in previous areas and reuse them. Most of the encoding time is spent in searching for similar patterns. The more complex and thorough search algorithms will take longer time. There are five methods presented here in order of complexity [43]:

• Diamond (dia): the simplest search pattern, it checks patterns at four directions (up, left, down, and right) then picks the best candidate.

• Hexagon (hex): works similarly to Diamond, but uses 6-point hexagon instead.

• Uneven multi-hexagon (uhm): similar to Hexagon, but it is able to avoid missing harder-to-find motion vectors.

• Exhaustive (esa): searches the complete motion vector space within a specified range.

• Transformed exhaustive ($tesa$): similar to exhaustive, but with additional complexities [43].

As seen in Table 4.11, all the search methods provide almost similar video quality and similar average frames size. Regarding latency, the two exhaustive methods perform poorly as they take double the time without any noticeable improvements in quality. The diamond method is the fastest in this scenario. However, we must stress on the fact that the performance of these methods depends highly on the video content.

Table 4.11: Motion Estimation Methods in H.264.

Search Pattern	Avg. Frame Latency (ms)	Avg. Frame Size (KB)	PSNR	SSIM
Diamond	66.32	71.03	44.33	14.42
Hexagon	69.18	71.36	44.31	14.44
UMH	73.18	71.41	44.36	14.46
Exhaustive	137.55	72.08	44.38	14.49
TESA	147.23	72.2	44.45	14.51

4.2.13 Lookahead ($-rc - lookahead$)

This setting specifies the number of frames to lookahead for macroblock tree rate control [44]. We notice once again that we have to make a trade off between quality, latency, and frame size. As can be seen in Table 4.12, a higher lookahead value will improve the quality, but it will slow the encoding process and increase the frame size. The default lookahead value is 40. If latency is more important than quality, a much slower value should be chosen. For example, using the value 2, we achieve lower latency and smaller frame size at the cost of lower quality.

Table 4.12: Lookahead Settings.

Setting	Avg. Frame Lat. (ms)	Avg. Frame Size (KB)	PSNR	SSIM
-rc-lookahead 2	61.31	53.86	43.6	14.02
-rc-lookahead 5	63.21	63.05	44.23	14.46
-rc-lookahead 10	64.8	68.86	44.24	14.33
Default setting	68.35	71.36	44.34	14.43

CHAPTER 5: CONCLUSION

In this work, we analyzed the latency of streaming video applications for edge computing. In particular, we analyzed the encoding latency, and the transmission and receive latencies in the Linux networking stack. We conclude that the encoding latency is much higher than the latency in the networking stack. Although the Linux networking stack has been blamed for its huge latency [31] [32]. However this does not apply for our work as we used UDP instead of TCP, and the packet sizes were rather large. Within the networking stack, the latency at the kernel and user boundary is a few times higher than the latency in the IP and UDP layers. However, the total latency is quite small compared to the encoding latency.

The combined Average transmission and receive latency in the networking stack is about 50 μs. While The average encoding latency per frame was around 70 ms. The encoding latency is comparably high. Applying the optimum parameters can reduce the encoding latency by 50%. The most notable reduction in latency occurs when B-frames are disabled. However, reducing the encoding latency often results in either degrading the quality or increasing the bitrate.

The encoding process is computationally intensive, so in order to achieve real-time encoding, hardware implementation is often required [35] [36]. Another solution that eliminates the encoding and decoding latencies altogether is using raw videos. However, this will require very high bitrate which makes it suitable for only small range of applications such as for application that share videos over a local area network.

For future work, the tail latency per frame can be measured for the encoding and decoding. In addition, finding the optimum combination of the encoding parameters can further reduce the encoding latency.

REFERENCES

[1] A. Khan, L. Sun, and E. Ifeachor, "Content clustering based video quality prediction model for mpeg4 video streaming over wireless networks," in *Communications, 2009. ICC'09. IEEE International Conference on*, pp. 1–5, IEEE, 2009.

[2] D. Z. Rodriguez, J. Abrahao, D. C. Begazo, R. L. Rosa, and G. Bressan, "Quality metric to assess video streaming service over tcp considering temporal location of pauses," *IEEE Transactions on Consumer Electronics*, vol. 58, no. 3, pp. 985–992, 2012.

[3] M.-y. Chen, L. Mummert, P. Pillai, A. Hauptmann, and R. Sukthankar, "Exploiting multi-level parallelism for low-latency activity recognition in streaming video," in *Proceedings of the first annual ACM SIGMM conference on Multimedia systems*, pp. 1–12, ACM, 2010.

[4] S. Wenger, "H. 264/avc over ip," *IEEE Transactions on circuits and systems for video technology*, vol. 13, no. 7, pp. 645–656, 2003.

[5] M. A. Lema, A. Laya, T. Mahmoodi, M. Cuevas, J. Sachs, J. Markendahl, and M. Dohler, "Business case and technology analysis for 5g low latency applications," *arXiv preprint arXiv:1703.09434*, 2017.

[6] M. Ghasemi, P. Kanuparthy, A. Mansy, T. Benson, and J. Rexford, "Performance characterization of a commercial video streaming service," in *Proceedings of the 2016 Internet Measurement Conference*, IMC '16, (New York, NY, USA), pp. 499–511, ACM, 2016.

[7] S. Yi, C. Li, and Q. Li, "A survey of fog computing: concepts, applications and issues," in *Proceedings of the 2015 Workshop on Mobile Big Data*, pp. 37–42, ACM, 2015.

[8] T. Bird, "Measuring function duration with ftrace," in *Proceedings of the Linux Symposium*, pp. 47–54, Citeseer, 2009.

[9] *STRACE(1) Linux User's Manual*, March 2010.

[10] *TCPDUMP(8) Linux User's Manual*, September 2015.

[11] *WIRESHARK(8) Linux User's Manual*, 2.0.2 ed., February 2016.

[12] "Systemtap documentaion." `http://sourceware.org/systemtap/`. [Online; accessed 27-April-2017].

[13] L. Kernel, "Linux ip sysctl documentation," *Documentation/networking/ip-sysctl. txt*, 2012.

[14] *ETHTOOL(8) Linux User's Manual*, 4.5 ed., March 2016.

[15] A. Durresi and R. Jain, "Rtp, rtcp, and rtsp - internet protocols for real-time multimedia communication.," in *The Industrial Information Technology Handbook*, CRC Press, 2005.

[16] A. R. Rind, K. Shahzad, and M. A. Qadir, "Evaluation and comparison of tcp and udp over wired-cum-wireless lan," in *Multitopic Conference, 2006. INMIC'06. IEEE*, pp. 337–342, IEEE, 2006.

[17] R. Nishtala, H. Fugal, S. Grimm, M. Kwiatkowski, H. Lee, H. C. Li, R. McElroy, M. Paleczny, D. Peek, P. Saab, *et al.*, "Scaling memcache at facebook.," in *nsdi*, vol. 13, pp. 385–398, 2013.

[18] N. Bahri, T. Grandpierre, M. A. B. Ayed, N. Masmoudi, and M. Akil, "Embedded real-time h264/avc high definition video encoder on tis keystone multicore dsp," *Journal of Signal Processing Systems*, vol. 86, no. 1, 2017.

[19] Y.-K. Lin, C.-C. Lin, T.-Y. Kuo, and T.-S. Chang, "A hardware-efficient h. 264/avc motion-estimation design for high-definition video," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 55, no. 6, pp. 1526–1535, 2008.

[20] M. Moghaddam and C. Ababei, "Performance evaluation of network-on-chip based h. 264 video decoders via full system simulation," *IEEE Embedded Systems Letters*, 2017.

[21] Y. V. Ivanov and C. J. Bleakley, "Real-time h. 264 video encoding in software with fast mode decision and dynamic complexity control," *ACM Transactions on Multimedia Computing, Communications, and Applications (TOMM)*, vol. 6, no. 1, p. 5, 2010.

[22] D. Wu, Z. Xue, and J. He, "icloudaccess: Cost-effective streaming of video games from the cloud with low latency," *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 24, no. 8, pp. 1405–1416, 2014.

[23] C. Benvenuti, *Understanding Linux network internals*. " O'Reilly Media, Inc.", 2006.

[24] A. K. Chimata, "Path of a packet in the linux kernel stack," *University of Kansas*, 2005.

[25] R. Rosen, "Wireless linux kernel networking-advanced topics," 2009.

[26] "Networking:kernel flow." `https://wiki.linuxfoundation.org/networking/kernel_flow`. [Online; accessed 28-April-2017].

[27] H. Glenn, *Linux IP Networking: A guide to the implementation and modification of the Linux Protocol Stack*. PhD thesis, Masters Thesis http://www. cs. unh. edu/cnrg/gherrin/linuxnet. html, 2000.

[28] "Linux networking stack from the ground up." https://www.privateinternetaccess.com/blog/2016/01/linux-networking-stack-from-the-ground-up-part-1/. [Online; accessed 25-April-2017].

[29] D. Siemon, "Queueing in the linux network stack," *Linux Journal*, vol. 2013, no. 231, p. 2, 2013.

[30] M. E. Haque, Y. He, S. Elnikety, R. Bianchini, K. S. McKinley, *et al.*, "Few-to-many: Incremental parallelism for reducing tail latency in interactive services," *ACM SIGPLAN Notices*, vol. 50, no. 4, pp. 161–175, 2015.

[31] K. Yasukata, M. Honda, D. Santry, and L. Eggert, "Stackmap: Low-latency networking with the os stack and dedicated nics," in *2016 USENIX Annual Technical Conference (USENIX ATC 16), Denver, CO*, 2016.

[32] G. Chuanxiong and Z. Shaoren, "Analysis and evaluation of the tcp/ip protocol stack of linux," in *Communication Technology Proceedings, 2000. WCC-ICCT 2000. International Conference on*, vol. 1, pp. 444–453, IEEE, 2000.

[33] J. S. Chase, A. J. Gallatin, and K. G. Yocum, "End system optimizations for high-speed tcp," *IEEE Communications Magazine*, vol. 39, no. 4, pp. 68–74, 2001.

[34] M. Majkowski, "How to receive a million packets per second." https://blog.cloudflare.com/how-to-receive-a-million-packets/. [16-June-2015].

[35] T.-C. Wang, Y.-W. Huang, H.-C. Fang, and L.-G. Chen, "Performance analysis of hardware oriented algorithm modifications in h. 264," in *Acoustics, Speech, and Signal Processing, 2003. Proceedings.(ICASSP'03). 2003 IEEE International Conference on*, vol. 2, pp. II–493, IEEE, 2003.

[36] T.-C. Chen, C. Lian Jr, and L.-G. Chen, "Hardware architecture design of an h. 264/avc video codec," in *Proceedings of the 2006 Asia and South Pacific Design Automation Conference*, pp. 750–757, IEEE Press, 2006.

[37] "ffmpeg documentation." https://ffmpeg.org/. [Online; accessed 26-April-2017].

[38] "Low-latency live streaming your desktop using ffmpeg." http://fomori.org/blog/?p=1213. [02-August-2015].

[39] I. E. Richardson, *The H. 264 advanced video compression standard*. John Wiley & Sons, 2011.

[40] Z. Kotevski and P. Mitrevski, "Experimental comparison of psnr and ssim metrics for video quality estimation," in *ICT Innovations 2009*, pp. 357–366, Springer, 2010.

[41] W. Robitza, "Crf guide (constant rate factor in x264 and x265)." http://slhck.info/video/2017/02/24/crf-guide.html. [Online; accessed 28-April-2017].

[42] "Ffmpeg: Streaming guide." https://trac.ffmpeg.org/wiki/StreamingGuide. [Online; accessed 28-April-2017].

[43] J. Patterson, "Video encoding settings for h.264 excellence." http://www.lighterra.com/papers/videoencodingh264/. [Online; accessed 28-April-2017].

[44] "H.264 encoding guide." https://www.avidemux.org/admWiki/doku.php?id=tutorial:h.264. [Online; accessed 28-April-2017].

APPENDIX A: List of SystemTap Scripts

A.1 Measuring the Duration of the Interrupt Handler

```
// global varibales
global irq_enter; // an array to store the timestamp of the start of executing the interrupt
    handler
global irq_return;// an array to store the timestamp of the returning from the interrupt
    handler
global num; // The total number of interrupts
probe kernel.function("handle_irq")
{
q = $desc->irq_data->irq //To get the IRQ number
//probe only the interrupts associated with the ethernet interface
if ( (q == 55) || (q ==56) || (q ==57) || ( q ==58) | ( q ==59) || ( q == 60) ||( q ==
    61) || ( q == 62) || (q == 63))
        {
        num <<< 1;
        irq_enter[cpu()] = gettimeofday_ns();
        }
}
probe kernel.function("handle_irq").return // probe when the function returns
{
 q = $desc->irq_data->irq
if ( (q == 55) || (q ==56) || (q ==57) || ( q ==58) | ( q ==59) || ( q == 60) ||( q ==
    61) || ( q == 62) || (q == 63))
    {
      irq_return[cpu()] = gettimeofday_ns(); // time stamp the interrupt
      printf("%d\n", irq_return[cpu()] - irq_enter[cpu()]) // print the Latency
    }
```

```
}
probe timer.s(10) // count the number of interrupts
{
printf("===> The number of interrupts = %d \n", @count(num))
exit();
}
```

A.2 Measuring the Latency in the IP and UDP Layers

```
// global variab
global skb_copy% [100000];
global netif% [100000] ;
global id_ip;
probe kernel.function("ip_rcv")
{
    nh_addr = $skb->head + $skb->network_header; // the address of the beginning of
        the network header
    trans_addr =$skb->head + $skb->transport_header// the address of the beginning of
        the transport header
    d_port = ntohs(kernel_short(trans_addr+2)) // the destination port
    id_ip_1= ntohs(kernel_short(nh_addr+4)) // the packet id
    if (d_port == 3333) // filtering only the packets related to this experiment
    {
     netif[id_ip_1]= gettimeofday_ns(); // timestamping
    }
}


// Probe the entry of the enqueuing the packet into the socket
probe kernel.function("sock_queue_rcv_skb")
```

```
{
    nh_addr = $skb->head + $skb->network_header; // the address of the network
        header
    trans_addr =$skb->head + $skb->transport_header // the address of the transport
        header
    d_port = ntohs(kernel_short(trans_addr+2)) // the destination port
    id_ip= ntohs(kernel_short(nh_addr+4)) // the packet id
    if (d_port == 3333)
      {
       skb_copy[id_ip]= gettimeofday_ns(); // time stamp at the end of the quiuing the
            packet into the socket
      }
}
// Compute the latency and store the packets every 1 ms, so the packets' ids won't wrap
    around
probe timer.ms(1)
{
foreach ([id_ip] in skb_copy )
     {
      if ( skb_copy[id_ip] != 0) // if a time stamp has been taken
         {
         lat = skb_copy[id_ip] - netif[id_ip] ; // compute the latency for this particular
              packet
          printf("%d \n", lat)
         }
      }
delete skb_copy; // delete the array elements as we have alreay used them to find the latency


}
```

A.3 Measuring The Latency of Enqeueuing the Packet into the Socket

```
// global variables
global sock_queu;
global sock_q_ret;
global id_ip;

//probing the entry of the function
probe kernel.function("sock_queue_rcv_skb")
{
    nh_addr = $skb->head + $skb->network_header; // the address of the beginning of
        the network header
    trans_addr =$skb->head + $skb->transport_header// the address of the beginning of
        the transport header
    d_port = ntohs(kernel_short(trans_addr+2)) // the destination port
    id_ip= ntohs(kernel_short(nh_addr+4)) // the packet id
    if (d_port == 3333)
    {
     sock_queu[id_ip]= gettimeofday_ns(); //timestamping
    }
}

//probing the return from the function
probe kernel.function("sock_queue_rcv_skb").return
{
    nh_addr = $skb->head + $skb->network_header;
    trans_addr =$skb->head + $skb->transport_header
```

```
    d_port = ntohs(kernel_short(trans_addr+2))
    id_ip_1= ntohs(kernel_short(nh_addr+4))
    if (d_port == 3333)
    {
     sock_q_ret[id_ip_1]= gettimeofday_ns();
    }
}

// calculate the measured latency every 1 ms
probe timer.ms(1)
{
foreach ([id_ip] in sock_queu)
     {
      if ((sock_queu[id_ip]) != 0)
          {
           lat = sock_q_ret[id_ip] −sock_queu[id_ip]
           if (lat > 0)
                printf("%d\n", lat)
           }
      }
delete sock_queu;
delete sock_q_ret;
}
```

A.4 Measuring the Latency between Enqueuing the Packet into Socket and Delivering it to User Space

```
global skb_copy% [100000];
global udp_queue% [100000] ;
```

```
global id_ip;
// probing after finishing enquiing the packet in the socket queue
probe kernel.function("sock_queue_rcv_skb").return
{
    nh_addr = $skb->head + $skb->network_header; // getting a pointer to the IP
        header
    trans_addr =$skb->head + $skb->transport_header // getting a pointer to the
        transport header
    d_port = ntohs(kernel_short(trans_addr+2)) // the destination port from the udp
        header
    id_ip_1= ntohs(kernel_short(nh_addr+4)) //the identification field in the ip header
    if (d_port == 3333)
    {
      udp_queue[id_ip_1]= gettimeofday_ns(); // Storing the time stamp in an array
          indexed by the packet id
    }
}
// probing delivering the packet to user space
probe kernel.function("skb_copy_datagram_iter")
{
    nh_addr = $skb->head + $skb->network_header; // Getting a pointer to the IP
        header
    trans_addr =$skb->head + $skb->transport_header // Getting a pointer to the
        transport header
    d_port = ntohs(kernel_short(trans_addr+2)) // the destination port from the udp
        header
    id_ip= ntohs(kernel_short(nh_addr+4)) //the identification field in the ip header
    if (d_port == 3333) // filtering only the packets related to this experiment
    {
```

```
      skb_copy[id_ip]= gettimeofday_ns(); // Storing the time stamp in an array indexed by
          the packet id
     }
}
// Compute the latencies and store the results evert 1 ms, so the packets' ids won't wrap
     around
probe timer.ms(1)
{
foreach ([id_ip] in skb_copy ) // iterate for each packet
      {
       if ( skb_copy[id_ip] != 0) // if the time stamp is already taken
          {
           lat = skb_copy[id_ip] - udp_queue[id_ip] ; // measure the latency
           printf("%d \n", lat)
          }
       }
delete skb_copy; // delete the elements of the array as we already used them to compute the
     latency
}
```

A.5 Measuring the Duration of $sys\_recvmsg()$

```
// probing the return of ___sys_recvmsg()
probe kernel.function("___sys_recvmsg").return
{
 num <<< 1 // a counter
 // computing the latency between the entry and exit of each call to the function
 printf("time = %d and the return value = %d\n", gettimeofday_ns() - @entry(
      gettimeofday_ns() ), $return )
}
```

A.6 Finding the Locations of Packet Drops

```
global drops
probe kernel.trace("kfree_skb") // the function that called to drop the packets
 {
        drops[$location] <<< 1 // count the number of dropped packetes
 }
probe timer.sec(5) // print the drop locations every 5 seconds
{
  foreach (i in drops−)
  {
    printf("%d packets dropped at %s\n", @count(drops[i]), symname(i))
  }
  delete drops
}
```

A.7 Measuring the Latency of the First Part of the Transmission Side

```
global sys_sendto;
global t, t1;

probe kernel.function("SyS_sendto")
{
  sys_sendto[cpu()] = gettimeofday_ns();
}

probe kernel.function("udp_send_skb")
{
  nh_addr = $skb−>head + $skb−>network_header; // getting a pointer to the IP header
```

```
  trans_addr =$skb->head + $skb->transport_header // getting a pointer to the
      transport header
  d_port = ntohs(kernel_short(trans_addr+2)) // the destination port from the udp header
  id_ip_1= ntohs(kernel_short(nh_addr+4)) //the identification field in the ip header
  if (d_port == 3333)
  {
    cpu = cpu()
    t[cpu] = gettimeofday_ns();
    t1 = t[cpu] - sys_sendto[cpu]
    printf("%d\n", t1);
  }
}
```

A.8 Measuring the Latency of the Second Part of the Transmission Side

```
global skb_copy% [100000]; // wrapping array to store the timestamping
global udp_queue% [100000] ;
global id_ip; // the packet id

probe kernel.function("udp_send_skb")
{
    nh_addr = $skb->head + $skb->network_header; // getting a pointer to the IP
        header
    trans_addr =$skb->head + $skb->transport_header // getting a pointer to the
        transport header
    d_port = ntohs(kernel_short(trans_addr+2)) // the destination port from the udp
        header
    id_ip_1= ntohs(kernel_short(nh_addr+4)) //the identification field in the ip header
    if (d_port == 3333)
    {
```

```
      udp_queue[id_ip_1]= gettimeofday_ns(); // Storing the time stamp in an array
          indexed by the packet id
      }
}

probe kernel.function("dev_hard_start_xmit")
{
      nh_addr = $first->head + $first->network_header;
      trans_addr =$first->head + $first->transport_header
      d_port = ntohs(kernel_short(trans_addr+2))
      id_ip= ntohs(kernel_short(nh_addr+4))
      if (d_port == 3333) // filtering only the packets related to this experiment
      {
          skb_copy[id_ip]= gettimeofday_ns();
      }
}

// Compute the latencies and store the results evert 1 ms, so the packets' ids won't wrap
      around
probe timer.ms(1)
{
foreach ([id_ip] in skb_copy ) // iterate for each packet
      {
       if ( skb_copy[id_ip] != 0) // if the time stamp is already taken
          if ( udp_queue[id_ip] != 0)
          {
                lat = skb_copy[id_ip] - udp_queue[id_ip] ; // measure the latency
                printf("%d \n", lat)
          }
```

```
        }
delete skb_copy; // delete the elements of the array as we already used them to compute the
    latency
  }
```

A.9 Measuring the Encoding and Packetization Latency

```
global entry


probe kernel.function("sys_execve")
{
entry[pid()] = gettimeofday_ms()
}


probe kernel.function("sys_exit_group")
{
latency = gettimeofday_ms() - entry[pid()]
  printf("Time in ms = %d for pid = %d and execname() = %s\n\n", latency, pid(),
      execname())
}
```