A New XDP-based WebRTC Protocol Stack

Zih-Hong Lin
Department of Computer Science and Engineering
National Chung Hsing University
Taichung, Taiwan, R.O.C.
benson060211@gmail.com

Hsung-Pin Chang
Department of Computer Science and Engineering
National Chung Hsing University
Taichung, Taiwan, R.O.C.
hpchang@cs.nchu.edu.tw

***Abstract*—Abstract—With the rapid development of network technology, people increasingly rely on WebRTC applications, resulting in exponential growth in network bandwidth. In modern operating systems, the design of the network stack prioritizes generality over efficiency in order to accommodate a wide range of hardwares. As a result, the existing architecture has gradually become a performance bottleneck. Therefore, this paper proposes a WebRTC protocol stack that integrates Linux XDP technology and improves existing WebRTC libraries, thereby offloading the processing of WebRTC packets from the conventional network stack to XDP programs and WebRTC libraries. Experimental results demonstrate that, compared to the existing architecture, the proposed approach improves packet processing efficiency by approximately 19.6**

***Index Terms*—WebRTC, network stack, XDP, Linux**

I. Introduction

In the post-pandemic era, remote education and telecommuting have become part of daily life, boosting the use of instant messaging and remote desktop applications. Most rely on WebRTC (Web Real-Time Communication) to transmit audio and video. Since WebRTC integrates multiple existing technologies and protocols, its growing traffic has made efficient resource allocation increasingly important.

Today's hosts mainly depend on the OS network stack, which prioritizes generality over efficiency and thus becomes a performance bottleneck. Solutions such as DPDK and XDP offload packet processing from the stack to reduce latency, but they are mostly used in servers or IoT, with little focus on real-time communication. High-end hardware is also impractical for ordinary users, creating a need for a "just right" solution that balances efficiency and cost.

This paper proposes a WebRTC protocol stack that bypasses the Linux network stack. By integrating XDP programs into the kernel and enhancing two open-source WebRTC transport libraries—Libdatachannel and Libjuice—packet processing is shifted to XDP and the libraries, benefiting all applications built on this architecture. The approach improves efficiency entirely through software, without requiring hardware upgrades. By determining which tasks should run in XDP and which in user space, the system achieves about a 19.6% gain in packet processing efficiency, offering a practical solution for legacy devices.

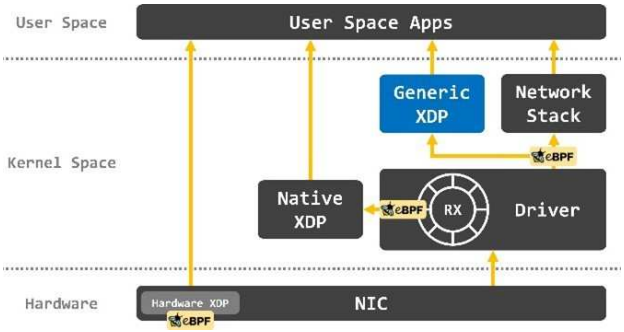

Fig. 1. The three XDP modes

II. Background

A. eXpress Data Path

XDP is a high-speed packet processing mechanism built on eBPF (extended Berkeley Packet Filter). eBPF allows small code snippets to be dynamically inserted into the Linux kernel, providing easier development and higher security than kernel modules.

Figure 1 (right) shows how the Linux OS processes packets. Depending on NIC support, XDP has three modes: Hardware, Native, and Generic.

As shown on the left side of Figure 1, Hardware XDP runs before packets enter the kernel space [9]. Because programs execute directly on the NIC, it offers the highest efficiency. However, only certain smart NICs support it, and both eBPF maps and helper functions are limited, restricting development [10].

Native XDP is triggered when packets enter the RX queue, and its operation depends on whether the net-work card driver provides support. At present, most high-speed network cards (10 Gbps and above) sup-ported in the Linux operating system include such functionality.

Generic XDP runs after packets leave the driver and before entering the network stack. It is the least efficient but requires no hardware support, making it the most flexible and cost-effective option.

Different XDP types fit different scenarios. Since most consumer devices lack high-end NICs, this paper uses Generic XDP. XDP programs can drop packets, pass them to the stack, or redirect them to user space. For redirection, packets go

through an AF_XDP socket (XSK), bypassing the stack. If the program does not modify the packet, user space receives a pointer to the Layer 2 header.

B. WebRTC

WebRTC is an open-source real-time communication technology based on a peer-to-peer (P2P) architecture. Most applications run as web apps and only require mainstream browsers. Instead, some WebRTC applications are still implemented as as native apps, where developers can use existing libraries or write their own code to interact with the OS. In order to enable WebRTC applications to receive packets from Generic XDP, this paper adopts such an architecture.

WebRTC libraries are mainly divided into three components: the voice engine, the video engine, and the transport module. This paper focuses on the transport component.

C. Libdatachannel

Libdatachannel is an open-source WebRTC transport library implemented in C++, designed to facilitate WebRTC connection-related procedures. The PeerConnection class defines numerous methods, and each time a PeerConnection instance is created, it is equivalent to initializing a corresponding WebRTC agent.

D. Libjuice

Libjuice is a submodule of Libdatachannel that handles ICE connection procedures. Because it relies on socket APIs and system calls, it is implemented in C for direct interaction with the Linux kernel.

Whenever a WebRTC agent enters the connection phase in preparation for ICE connectivity, it must first initialize a corresponding ICE agent. The Libjuice library accomplishes this initialization by creating an instance of the juice_agent structure. Once initialization is complete, the Libdatachannel library immediately requests candidates from it. The ICE agent first obtains the IP address of the network host and then creates a UDP socket to acquire a port number assigned by the Linux kernel. Together, the IP address and port constitute a host candidate.

III. System Implementation

A. System Architecture

In Figure 2, the yellow and green arrows show the receiving and transmitting paths of WebRTC packets. On the receiving path, packets are processed by Generic XDP; on the transmitting path, outgoing data is handled by the usual network stack. The blue blocks (WebRTC libraries and Generic XDP) mark the main areas of implementation and improvement in this paper.

B. Packet Receiving Path

1) Packet Interception: Figure 3 (right) shows the existing WebRTC packet workflow: the NIC driver and network stack process Layer 2–4 headers and deliver the sender's IP, port, and Layer 4 data to Libjuice via the socket API. In contrast,

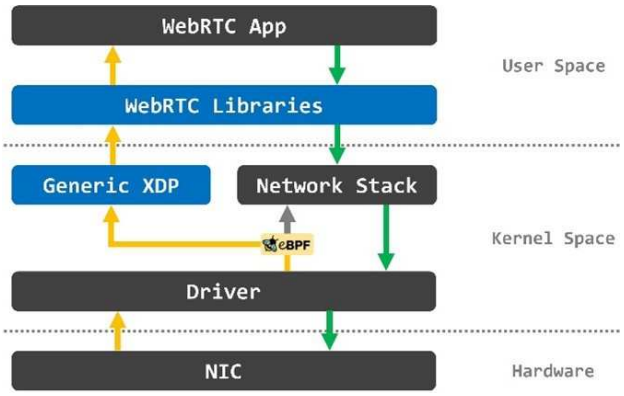

Fig. 2. System Architecture

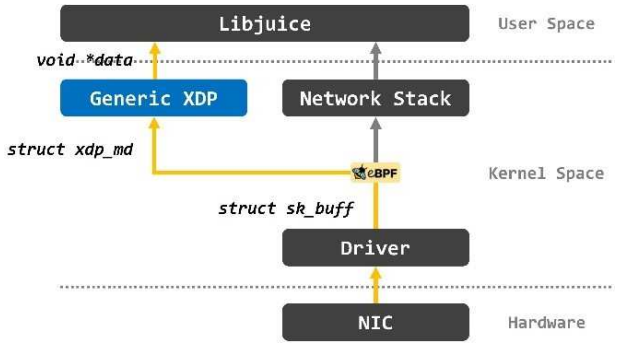

Fig. 3. WebRTC Packet Receiving Path with Generic XDP

with the proposed XDP program and eBPF hook attached to the NIC, all incoming Ethernet frames are intercepted and processed directly by XDP (Figure 3, left).

XDP does not directly access Ethernet frame contents; it obtains packet information via the xdp_md structure. The data and data_end pointers mark the frame's start and end. Using pointer arithmetic, an XDP program can parse headers and decide whether to redirect the frame to Libjuice. If redirection is chosen, the Libjuice library receives the starting address of the Ethernet frame (of type void *).

If the XDP program intercepts all Ethernet frames, how can other user space applications that rely on the existing architecture (i.e., the network stack) receive data?

2) Packet Classification: The XDP program implemented in this paper classifies all intercepted Ethernet frames into two catego-ries: those redirected to the Libjuice library and those passed back to the network stack.

To reduce processing time, the XDP program uses a two-stage classification: (1) check if the packet is UDP; if not, return it to the network stack; (2) if UDP, verify the destination port against Libjuice's port—redirect if it matches, otherwise return it to the stack.

At this point, a critical question arises: how does the XDP program determine the port currently being used by the Libjuice library?

When the XDP program implemented in this paper is loaded into the Linux kernel, two eBPF maps are created

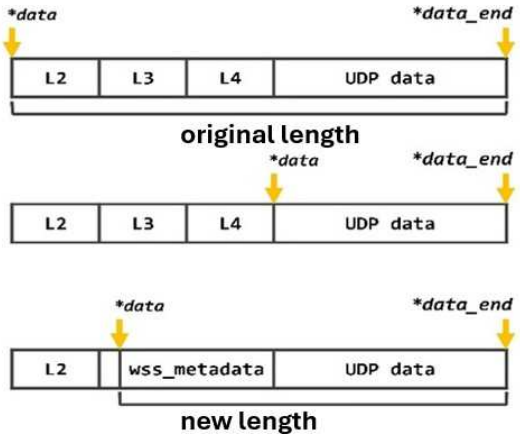

Fig. 4. Pointer Operations in the XDP Program

simultaneously:

- xsk_map: This map uses the RX queue index as the key and the corresponding XSK file descriptor as the value. Since a single NIC may have multiple RX queues, this design allows the XDP program to redirect Ethernet frames from different RX queues to their respective XSKs. In this paper, however, all RX queues of the NIC are mapped to the same XSK, which corresponds to the single XSK used in the Libjuice library.
- wss_map: This map uses the port number as the key and the corresponding recv_rb ad-dress as the value (with recv_rb explained in Section 3.2.4). Each time an ICE agent successfully establishes a UDP socket, a record is written, and the record is removed before the UDP socket is closed.

Suppose an ICE agent successfully establishes a UDP socket with port 9000. The ICE agent then writes its recv_rb address into wss_map[9000]. From that point on, whenever the XDP program implemented in this paper parses the destination port field, it uses the value to check whether a corresponding key exists in wss_map. All UDP packets with a destination port of 9000 are redirected to the Libjuice library's XSK, while the remaining packets are passed back to the network stack for processing

In the conn_poll mode, the number of entries in wss_map equals the number of UDP sockets currently used by the Libjuice library. Since eBPF maps store data in key-value pairs, the lookup time complexity remains O(1) regardless of the current connection scale.

3) Packet Content Simplification: In the implementation of XDP, the XSK only receives a pointer and a positive integer, corresponding to the starting address and length of the Ethernet frame, respectively, as shown in the first packet of Figure 4. This means that the Libjuice library must reparse the Layer 2 to Layer 4 headers on its own in order to obtain the required fields and data, which results in a lengthy packet processing workflow and unnecessary overhead. To address this, the XDP program implemented in this paper modifies the contents and the data pointer of Ethernet frames that are scheduled to be redirected to the XSK.

First, the XDP program sequentially parses the Layer 2 to Layer 4 headers to compute the starting address of the UDP data, and assigns this address to the data pointer, resulting in conn registry structure, of which only a single instance exists globally, serving as the manager for all ICE agents. This instance is dynamically allocated using the malloc() function, and its address is assigned to a pointer named registry. Its lifetime begins when the first ICE agent is initialized and ends when the last ICE agent is released. Each time an ICE agent is initialized, its address must be registered in the registry, and all ICE agents also store the address of the registry, forming a bidirectional association.

To enable the Libjuice library to receive packets from the XDP program via the XSK, this paper defines an xdp_info structure, of which only a single instance exists globally. It is dynamically allocated using the malloc() function when the registry is created, and its address is assigned to a pointer named juice_xdp (one of the member variables of registry). Its lifetime begins when the registry is created and ends when the registry is released, forming a unidirectional composition relationship between the two. When juice_xdp is created, its initialization procedure is as follows:

1) Obtain the file descriptor of xsk_map: Since Step 5 will write data into xsk_map, the XDP program implemented in this paper must first be loaded into the Linux kernel in order to successfully obtain its file descriptor.
2) Obtain the file descriptor of wss_map: As described in Section 3.2.2, the contents of wss_map must be updated whenever a UDP socket is created or closed.
3) Create UMEM: UMEM is a shared memory region allocated in user space, used to temporarily store the contents of packets redirected by the XDP program.
4) Create the XSK: The xsk_socket_create() function is used to create the XSK. Once created, its file descriptor is obtained, and its address is assigned to a pointer named xsk.
5) Bind the XSK to all RX queues: As described in Section 3.2.2, the indices of all RX queues of the NIC and the file descriptor of the XSK are written into xsk_map. After binding is completed, the XDP program can identify the redirection target and write the packet contents into the UMEM corresponding to the XSK.
6) Write all available addresses into the fill ring: The fill ring is a Single-Producer-Single-Consumer (SPSC) ring buffer located within the UMEM, used to record the currently available UMEM addresses for the XDP program to write into. Since the UMEM has just been initialized and no regions have yet been written by the XDP program, all regions are in the available state. Therefore, in this step, the fill ring is filled all at once.

When a WebRTC application establishes two WebRTC connections, the architecture shown in Figure 5 is formed. The Libdatachannel library creates two PeerConnection instances

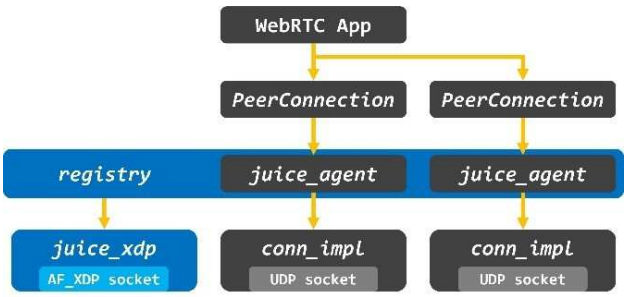

Fig. 5. Architecture with Two WebRTC Connections

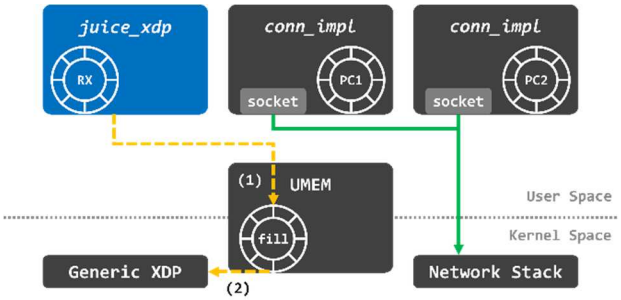

Fig. 6. Packet Transmission Path and the Read/Write Process of the Fill Ring

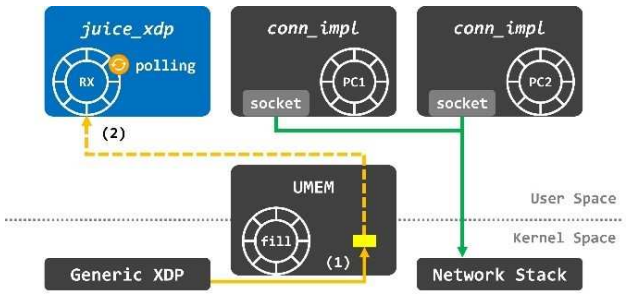

Fig. 7. Process of juice_xdp Obtaining Packet Contents from the RX Ring

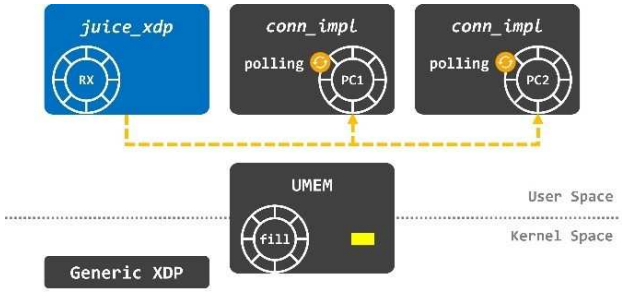

Fig. 8. Process of juice_xdp Distributing Packets to ICE Agents

(i.e., WebRTC agents), each of which owns a juice_agent instance (i.e., ICE agent). Since the Libjuice library adopts the conn_poll mode, each ICE agent has its own UDP socket, which is managed by a conn_impl structure.

The green solid arrows in Figure 6 represent the packet transmission path. This paper retains the existing architecture, where packets are transmitted through their respective UDP sockets. However, in the case of packet reception, all packets are first received collectively by the juice_xdp implemented in this paper and are then further distributed to the corresponding ICE agents.

Whenever there is an available address in the UMEM, juice_xdp writes the address into the fill ring (yellow dashed arrow (1) in Figure 6). Before the XDP program can write packet contents into the UMEM, it must first read an available address from the fill ring (yellow dashed arrow (2) in Figure 6). If the fill ring is empty, it indicates that no region is currently available for writing. Therefore, the producer and consumer of the fill ring are juice_xdp and the XDP program, respectively. After the XDP program reads an available address from the fill ring, it writes the packet contents to that address (yellow solid arrow (1) in Figure 7). Once the write is completed, the XDP program writes the ad-dress into the RX ring belonging to juice_xdp (yellow dashed arrow (2) in Figure 7). The RX ring is also an SPSC ring buffer, monitored by a polling thread to check for new contents. When the polling thread detects readable data in the RX ring, it calls the relevant functions of juice_xdp to perform packet reception and subsequent distribution. Therefore, the producer and consumer of the RX ring are the XDP program and juice_xdp, respectively.

On the other hand, to receive packets distributed by juice_xdp, this paper defines an SPSC ring buffer named recv_rb for each ICE agent (PC1 and PC2 in Figure 8), replacing the original design in which UDP data was received directly through the UDP socket. However, when multiple ICE agents exist simultaneously, how does juice_xdp determine to which recv_rb the received packet should be distributed?

When juice_xdp obtains an address from the RX ring, the packet at that address has the format shown in the third packet of Figure 4. In the wss_metadata block, there is a field named xdp_agent_rb_ptr, which stores the address of the corresponding recv_rb. Through this field, juice_xdp can determine, via pointer operations, to which ICE agent's recv_rb the packet should be distributed.

In the original Libjuice library, conn_impl returns the sender information (of type addr_record structure) and the UDP data (of type char *) to the ICE agent. To preserve the original architecture, juice_xdp fills part of the wss_metadata fields into addr_record, and sets the data pointer to the starting address of the UDP data before returning it (by writing into recv_rb). The original design, in which the polling thread monitored the UDP socket, is no longer used; instead, it now monitors the RX ring and each recv_rb, for a total of (number of ICE agents + 1) targets. Therefore, it can be concluded that the producer of each rec_rb is juice_xdp, while the consumer is the corresponding ICE agent.

As for why the addr_record is not inserted in front of the UDP data during the XDP program stage, but instead the wss_metadata is inserted, there are two main considerations: First, the headroom area at the beginning of the packet is limited, and inserting the addr_record would cause the XDP program to fail the verification of the eBPF Verifier.

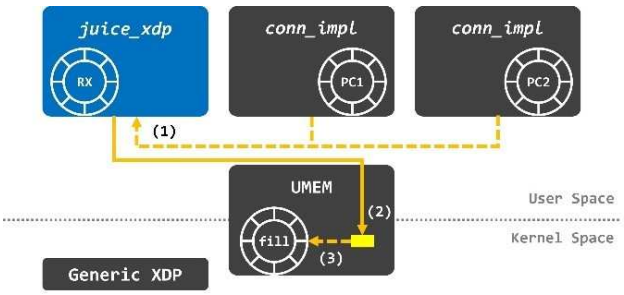

Fig. 9. Process of juice_xdp Recycling UMEM Addresses

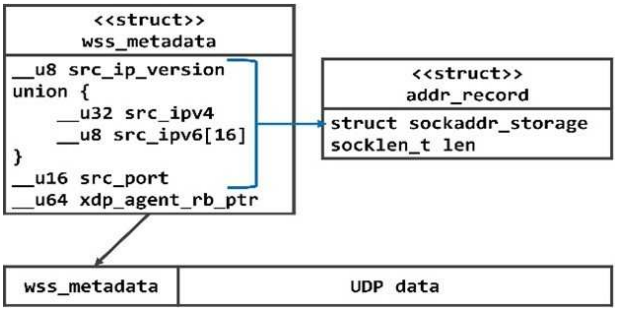

Fig. 10. Figure 10. Member Variables of wss_metadata and addr_record

Second, most fields in the addr_record are unused, and directly inserting it would result in significant space waste.

After the upper-layer application finishes processing the UDP data, conn_impl notifies juice_xdp that the address has been used (yellow dashed arrow (1) in Figure 9). Next, juice_xdp performs a recycling operation on the address, marking it as available again (yellow dashed arrow (2) in Figure 9), and then writes the address back into the fill ring to provide the XDP program with space for writing new packet contents (yellow dashed arrow (3) in Figure 9). At this point, the process returns to the initial state shown in Figure 6.

It is worth noting that in Figures 6 to 9, the solid ar-rows represent the access of packet data, while the dashed arrows represent the access of memory ad-dresses. To minimize the overhead of data copying, most of the processes rely on pointer operations to improve packet processing efficiency.

To maintain a modular and low-coupling design, most of the XSK-related implementations are concentrated in the two files xsk.c and xsk.h. Only a small number of existing files (such as conn.h, which declares conn_registry) require modifications, and these are conditionally compiled using the CMake flag USE_XDP to minimize the impact on the original logic of the Libjuice library.

C. XDP Program Implementation

The XDP program mainly relies on pointer operations to parse packet contents. However, due to the strict verification mechanism of the eBPF Verifier, every pointer operation must guarantee that it will not cause illegal memory access. For example, before accessing the IPv4 header, it is necessary to check whether the data pointer plus the IPv4 header length is less than the data_end pointer. Only if this condition is satisfied can the XDP program legally access that memory region; otherwise, the eBPF Verifier will reject the loading of the XDP program.

The packet in Figure 10 has the same format as the third packet in Figure 4, which is prepared for redirection to the Libjuice library. In the wss_metadata block, the types used are not the common C language types such as short, int, or long, but rather the fixed-width integer types defined by the Linux kernel. The reason is that the actual size of traditional types may vary across different architectures. For example, the long type occupies 8 bytes in a 64-bit environment but only 4 bytes in a 32-bit environment. To ensure that each variable maintains consistent size and alignment across different architectures, Linux kernel-level code commonly uses fixed-width types.

When the XDP program determines that an Ether-net frame should be redirected to the Libjuice library, it writes the queried recv_rb address into the xdp_agent_rb_ptr field of wss_metadata, to be used later by juice_xdp for distribution. To avoid errors in recording the recv_rb address caused by differences in memory address formats and sizes across architectures, the Libjuice library must first cast the address to the uintptr_t type before writing it into wss_map, ensuring that the original address can be correctly represented. It is then cast to the _u64 type to match the type definition of the xdp_agent_rb_ptr field.

As for the remaining fields in wss_metadata, they must be obtained by parsing the headers of the Ethernet frame. The fields enclosed by the blue brackets in Figure 10 are all intended to provide the necessary information for the Libjuice library to construct the addr_record. Among them, the sender's IP address is represented as a union structure, which contains fields for both IPv4 and IPv6 protocols to ensure correct recording.

If the Layer 3 protocol of the Ethernet frame is IPv4, the XDP program must first parse the header length field to calculate the correct header length, and then perform pointer operations based on this length. If the result is less than 20 bytes or greater than 60 bytes, it is considered invalid and the packet is immediately dropped. If the protocol is IPv6, since its header length is fixed at 40 bytes, the issue of variable length does not need to be considered.

D. The New Protocal Stack

The WebRTC protocol stack proposed in this paper aims to offload the processing of UDP packets from the network stack to the XDP program and the Libjuice library. The details are shown in Table 1.

Since the main traffic of WebRTC applications consists of multimedia messages, they can generally tolerate the loss of individual packets or short-term interruptions in audio and video without significantly affecting the user experience. Therefore, in pursuit of overall efficiency, this protocol stack ignores the UDP checksum and length verification to minimize packet processing time as much as possible.

TABLE I
PACKET PROCESSING TASKS OF THE PROPOSED PROTOCOL STACK AND THEIR OFFLOAD TARGETS

Protocol	Task	Offload Target
IPv4	Check header length	XDP program
	Check total length	–
	IPv4 checksum	XDP program
	Deliver to corresponding agent	Libjuice library
IPv6	Check header length	XDP program
	Check total length	–
	UDP checksum	–
	Deliver to corresponding agent	Libjuice library

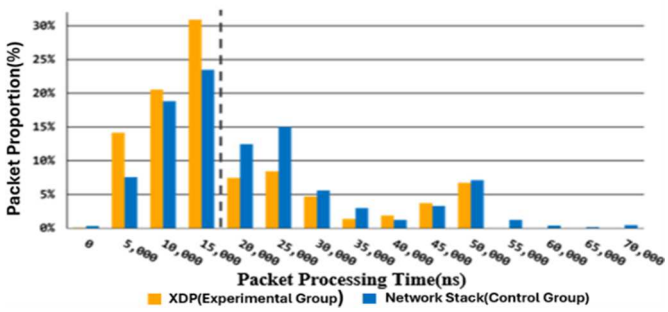

Fig. 11. Histogram of Packet Processing Time Distribution for the Experimental and Control Groups

IV. EXPERIMENTAL RESULTS

In this section, a WebRTC screen-sharing application is implemented. By measuring packet processing time, the performance differences between the architecture proposed in this paper and the existing architecture (using the Linux network stack) are compared.

The application consists of a sender and a receiver, where the sender transmits its screen display, and the receiver is responsible for receiving and rendering the display. Since this paper does not involve the architecture of packet transmission, the sender is implemented as a web application consisting only of an HTML file with embedded JavaScript, hosted on localhost:8000 and executed using the Firefox browser. The system resources required for screen transmission are allocated by the browser, and even under different hardware configurations, the allocated network bandwidth and performance are generally consistent, having no direct correlation with the hardware specifications.

The receiver is implemented as a native application written in C++, using the GStreamer library as the voice engine and video engine. As described in Section III.B.4, the architecture proposed in this paper introduces a CMake flag named USE_XDP, which allows switching between Libdatachannel and Libjuice libraries during the build stage: if US_XDP is set to 1, the proposed protocol stack is used to process packets; if USE_XDP is set to 0, the Linux network stack continues to be used for packet processing. This design not only simplifies the testing process but also ensures that, with identical upper-layer code, a fair comparison of packet processing time between the two methods can be achieved.

TABLE II
THE EXPERIMENTAL RESULTS.

	Experimental Group	Control Group
Maximum	54,053 ns	73,709 ns
Q3 (P75)	30,045 ns	39,715 ns
Median (P50)	18,325 ns	23,029 ns
Q1 (P25)	13,293 ns	15,590 ns
Minimum	4,068 ns	2,424 ns

This experiment executed the application three times, recording the timestamps of the first 500 packets in each run. Table 2 presents the results after removing outliers, with 1,378 valid data points retained for the experimental group and 1,308 for the control group (out of the original 1,500)

Figure 11 shows the results of grouping the data into intervals of 5,000 ns. Each interval in the figure represents the proportion of packets whose processing time falls within a specific range. For example, the interval of 15,000 ns indicates packets with processing times between 15,001 and 20,000 ns, where our XDP-based network stack and the original Linux network stack account for 30.91% and 23.47% of the total packets, respectively. To determine the proportion of packets with processing times below 20,000 ns, the intervals to the left of the dashed line are summed, yielding 65.67% for our XDP-based solution and 50.16% for the Linux network stack. This reflects that the packet processing times of our XDP-based solution are more concentrated in the low-latency range. In terms of tail latency, nearly 2% of the packets in the Linux network stack are distributed above 55,000 ns.

V. CONCLUSION

This paper proposed a WebRTC protocol stack combined with Linux XDP technology, enabling WebRTC packets to bypass the Linux network stack, be initially parsed directly by the XDP program, and then distributed by the Libjuice library in user space. This method improves packet processing efficiency by 19.6%.

REFERENCES

[1] "Webrtc for the curious." https://webrtcforthecurious.com/.
[2] N. Blum, S. Lachapelle, and H. Alvestrand, "Webrtc - realtime communication for the open web platform: What was once a way to bring audio and video to the web has expanded into more use cases we could ever imagine," Queue, vol. 19, no. 1, pp. 77–93, 2021.
[3] "What is ebpf? an introduction and deep dive into the ebpf technology." https://ebpf.io/what-is-ebpf/.
[4] T. Høiland-Jørgensen, J. D. Brouer, D. Borkmann, J. Fastabend, T. Herbert, D. Ahern, and D. Miller, "The express data path: fast programmable packet processing in the operating system kernel," 2018.
[5] "Github - paullouisageneau/ libdatachannel: C/C++ webrtc network library featuring data channels, media transport, and websockets." https://github.com/paullouisageneau/libdatachannel.
[6] "Github - paullouisageneau/libjuice: Juice is a udp interactive connectivity establishment library." https://github.com/paullouisageneau/libjuice.
[7] T. Le´vai, B. E. Kreith, and G. Re´tva´ri, "Supercharge webrtc: Accelerate turn services with ebpf/xdp," 2023