Janus Streamer: A Scalable WebRTC Media Streaming Server

1st Jaehyeong Park
Seoul National University
Seoul, Republic of Korea
jh_park@snu.ac.kr

2nd Seongyeop Jeong
Seoul National University
Seoul, Republic of Korea
seongyeop.jeong@snu.ac.kr

3rd Seung Min Kim
Fadu Inc.
Seoul, Republic of Korea
andykim@fadutec.com

4th Jin-Soo Kim
Seoul National University
Seoul, Republic of Korea
jinsoo.kim@snu.ac.kr

2026 IEEE 23rd Consumer Communications & Networking Conference (CCNC) | 979-8-3315-9673-6/26/$31.00 ©2026 IEEE | DOI: 10.1109/CCNC65079.2026.11366482

***Abstract*—Janus Gateway supports WebRTC streaming for real-time video and audio delivery over the Web, but suffers from scalability limitations. It adopts a modular architecture in which the core and plugins operate on separate threads. This design leads to performance degradation, as packets must traverse queues from the plugin threads to the core thread. To mitigate this inefficiency, we propose Janus Streamer, an enhanced streaming system that improves Janus Gateway's producer-consumer model. Janus Streamer provides a direct packet transmission interface for plugins and optimizes it to enhance performance. Experimental results show that Janus Streamer achieves a 3.6x improvement in throughput and a 37% reduction in CPU usage, while maintaining comparable Quality of Experience (QoE). Furthermore, it outperforms Janus Gateway on a single CPU core, achieving throughput improvements ranging from 3.1x to 5.7x across varying media bitrates. These findings underscore Janus Streamer's capacity to support large-scale media streaming, and suggest techniques broadly applicable to other media transmission systems.**

***Index Terms*—WebRTC, Janus Gateway, Streaming, UDP GSO**

I. Introduction

WebRTC (Web Real-Time Communication) [1] is a protocol suite designed to facilitate the real-time communication of video, audio, and data over the Web. The IETF (Internet Engineering Task Force) [2] has standardized the protocols, and the W3C (World Wide Web Consortium) [3] has defined the corresponding APIs. Using WebRTC, developers can implement real-time client applications directly in modern web browsers without additional client plugins. This capability makes it an attractive solution for real-time services, especially when transitioning existing applications to the Web. However, to leverage the WebRTC ecosystem, servers must implement WebRTC protocols. This task can be complex and resource-intensive when integrated into legacy infrastructure. In such scenarios, Janus Gateway functions as an effective solution.

Janus Gateway is a well-known, general-purpose, open-source WebRTC gateway [4]. It is built with a flexible architecture that allows plugins to be attached for handling specific tasks, making it highly adaptable to various use cases [5]. Out of the box, it offers basic plugins for tasks such as echo test, video calls, and streaming, which can be further customized to meet specific application needs. Janus Gateway has been utilized in many real-world applications such as telemedicine platforms, video conferencing, and interactive online education systems [6].

Janus Gateway also functions as a streaming server. Written in C, it is a high-performance and lightweight solution, making it ideal not only as a WebRTC gateway but also as a server for real-time media delivery. Similar to how Dolby Millicast [7] provides streaming services via WebRTC, Janus Gateway can be used to deliver comparable services. With support from the WebRTC ecosystem, developers can easily build streaming applications using Janus Gateway, taking advantage of its flexible plugin architecture to customize and extend functionality as needed.

However, one limitation of the existing Janus Gateway is its difficulty in handling a large number of connection states simultaneously. In our experience with a High Definition (HD) video/audio live streaming task using Janus Gateway, we found that the gateway struggled to manage the workload when several thousand of connections were established. This issue underscores a key challenge in scaling Janus Gateway-based services, particularly for large-scale live streaming. While Janus Gateway offers numerous advantages, addressing its limitations in handling large-scale connections is essential for expanding its use in more demanding applications.

This paper focuses on the performance issues that arise from the strictly separated producer-consumer model between the core and plugins in the Janus Gateway. To support extensible plugins, Janus Gateway is designed with a modular architecture that separates the core and plugins into distinct components, each operating on separate threads.

While this design helps ensure timely packet reception, it incurs substantial context-switching overhead as the number of threads increases. Additionally, it introduces overhead from queue manipulation and memory management, as packets are relayed between threads through queues. These structural inefficiencies in Janus Gateway may go unnoticed with a small number of clients or low CPU utilization but become increasingly significant as the client count grows.

We propose a new streaming solution based on the Janus Gateway, called Janus Streamer, which addresses the structural issues of the original design. The main idea of Janus Streamer is to enable plugin threads to transmit data packets directly, without invoking the core thread.

By eliminating the overheads from the producer-consumer

model, this approach speeds up packet transmission from the server to the clients. In addition, Janus Streamer improves performance by using `sendmmsg`, Generic Segmentation Offload (GSO) and CPU affinity. Despite these enhancements, Janus Streamer retains the fundamental architecture of the Janus Gateway, including its modularity, ensuring full backward compatibility with existing applications and infrastructure.

We demonstrate that Janus Streamer offers substantial improvements over Janus Gateway by evaluating both the server-side performance and client-side QoE (Quality of Experience). We found that Janus Streamer delivers a 3.6x throughput improvement and reduces CPU usage by 37%, while maintaining a similar level of QoE compared to Janus Gateway with streaming plugin.

The remainder of this paper is organized as follows. Section 2 discusses the architecture of Janus Gateway and highlights its scalability issues. Section 3 provides a detailed explanation of Janus Streamer structure. Section 4 outlines the methodology used to evaluate Janus Streamer's performance, followed by the results, and Section 5 concludes the paper.

II. Background and Motivation

A. Janus Gateway

Janus Gateway is a well-studied, high-performance, open-source WebRTC server that provides a solid foundation for scalable media streaming systems. While several commercial WebRTC servers exist (e.g., Ant Media [8], LiveSwitch [9], and Dolby [7]), their performance has not been publicly evaluated or compared. Consequently, prior studies have focused on open-source systems such as Janus Gateway [4], Kurento [10], Jitsi [11], Medooze [12], and Mediasoup [13]. André et al. [14] conducted a comparative analysis of WebRTC Selective Forwarding Unit (SFU) performance, demonstrating that Janus Gateway achieves the highest image quality and maximum bitrate under high-load conditions. While Janus Gateway exhibits higher latency with a small number of participants, its performance is comparable to Mediasoup and Medooze in scenarios involving over 150 participants. These observations make Janus Gateway particularly well-suited for developing high-performance WebRTC server functionalities, as evidenced by its extensive adoption in recent studies [15]–[17].

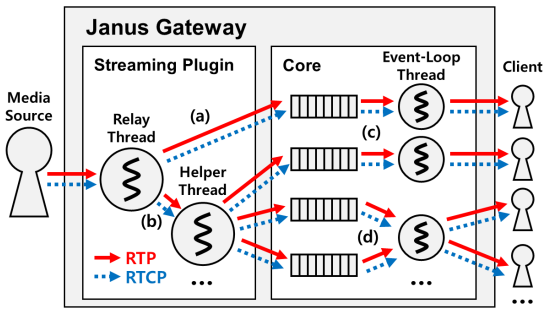


Fig. 1: Architecture of Janus Gateway

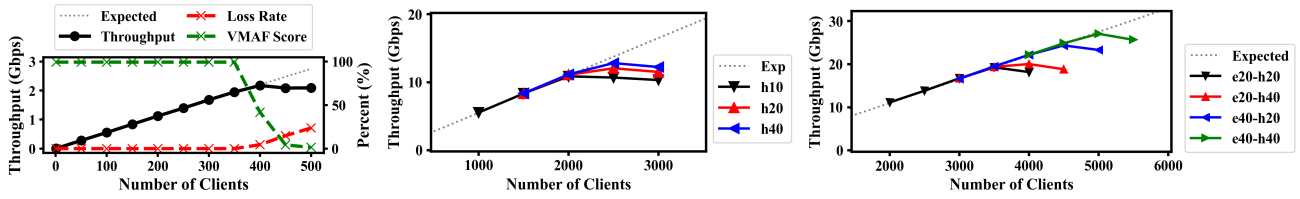
Janus Gateway utilizes a modular architecture that separates the plugins from the core, allowing for flexibility and extensibility in real-time communication systems. Figure 1 shows the overall architecture of Janus Gateway, including the *streaming plugin*. The system is equipped with a series of plugins that provide specialized capabilities and extend its functionality. These plugins are supported by the core that implements the WebRTC protocol and other essential features.

Plugins are designed to provide specific APIs and execute actions corresponding to their designated functions. Users interact with these plugins via APIs to request various functionalities. For example, the streaming plugin offers *start* and *stop* start and stop APIs for subscribing or unsubscribing to streams. The plugin is responsible for managing its associated user states and generating RTP (Real-time Transport Protocol) [18] packets, as well as the related RTCP (Real-time Transport Control Protocol) [18] packets. During live streaming operations, the streaming plugin receives incoming Video/Audio RTP packets, applies necessary modifications to RTP extensions, and enqueues them into each viewer's handle queue. Figure 1a illustrates the default packet relaying path, where the relay thread passes RTP and RTCP packets to the client handle queue.

Relaying packets by a single relay thread can create a bottleneck when the number of viewers increases. To rectify this bottleneck, starting from Janus Gateway v0.4.4, the streaming plugin introduces preconfigured helper threads designed to relay packets in parallel to multiple streaming viewers [4]. Figure 1b illustrates the relay thread transferring packets to a helper thread, which then forwards them to each client handle queue. Although the figure shows a single helper thread, utilizing multiple helper threads in practice allows for parallel processing across CPU cores, improving the relay throughput.

The central component, or the core, of Janus Gateway is responsible for managing communication with clients and processing events from the various plugins. The primary function of Janus Gateway is to relay packets to clients following the WebRTC protocol. According to the WebRTC protocol [19], clients communicate with servers or other clients via the `RTCPeerConnection` API, and Janus Gateway similarly implements its own `PeerConnection` structure. Before creating a `PeerConnection`, Janus Gateway must determine what kind of plugin functionality is required by each client. To handle this, Janus Gateway uses the session and handle structures. Each client is served by a dedicated session, with handles attached to the session to determine which plugins to use. A `PeerConnection` can then be created based on the requests of each plugin handle (e.g., initiating a stream). Requests and responses from clients regarding sessions and handles are exchanged via ad-hoc protocols, such as REST API, WebSocket, or RabbitMQ. Once a WebRTC `PeerConnection` is established, the core is responsible for transmitting RTP and RTCP packets from the plugins through this connection.

The core and plugins follow a producer-consumer model and run on separate threads. Plugin threads produce packets,



(a) Without any options (b) With helper (h) options (c) With static event-loop (e) and helper (h) options

Fig. 2: Scalability issues of Janus Gateway in 5 Mbps HD video/audio streaming on 40 CPU cores

while event-loop threads consume them, with packets relayed via a queue. Each plugin handle maintains its own queue; the event-loop thread dequeues packets or events from these queues and forwards them for further processing.

Janus Gateway supports two types of event-loop threads: per-handle and static. By default, a dedicated event-loop thread is created for each handle, as shown in Figure 1c. Alternatively, a fixed set of static event-loop threads can be configured to manage multiple handles. Figure 1d illustrates this configuration, where two clients each maintain their own handle queue, but a single static event-loop thread services both queues.

A key aspect of Janus Gateway's design is that plugins are solely responsible for generating and forwarding RTP/RTCP packets to the core. The core, in turn, encrypts these packets using the SRTP (Secured Real-time Transport Protocol) [20], which are subsequently transmitted to the client. This clear separation of responsibilities embodies the modular architecture of Janus Gateway, allowing plugin developers to focus solely on the creation and management of RTP packets without worrying about the operational complexities of the core.

B. Scalability Issue

We observed a significant throughput limitation during HD video and audio live streaming with Janus Gateway using the streaming plugin. As shown in Figure 2a, packet loss increases, and throughput saturates as the number of clients grows under the default configuration. The detailed experimental methodology is described in Section 4.1. Janus Gateway performs reliably with up to 350 clients, maintaining the expected throughput without any issues. However, when scaled to 500 clients, the throughput falls short of the expected 2.5 Gbps, reaching only around 2.0 Gbps, with a packet loss rate of 20%.

One primary cause of this limitation is that the streaming plugin relays packets to each media viewer using a single thread. To mitigate this limitation, we parallelized the packet relay process by enabling the helper thread option in the streaming plugin. In Figure 2b, the labels h10, h20, and h40 represent configurations where the number of helper threads is set to 10, 20, and 40, respectively. With these modifications, the system achieves a sustained throughput of 10 Gbps for up to 2,000 clients, but throughput saturates beyond this point. Although this enhancement supports over 5x clients than the default configuration, it only utilizes 10% of the available 100 Gbps network bandwidth.

Another scalability challenge arises from the Janus Gateway core, which becomes a bottleneck when managing a large number of connections. By default, Janus Gateway spawns a new thread for each client, leading to excessive context switching and degraded performance. To mitigate this, we implemented a static event-loop model, where a fixed number of threads handle shared event-loops across clients, reducing context-switching overhead. In Figure 2c, the label format *eX-hY* denotes configurations with X static event-loop threads and Y helper threads. With 40 static event-loop threads and 40 helper threads, the system supports up to 5,000 clients, achieving a throughput of 26 Gbps. However, this still utilizes only 26% of the available 100 Gbps NIC bandwidth, indicating persistent scalability limitations.

C. Motivation

Our research is motivated by the observation that simply modifying the plugin and adjusting its options is insufficient to resolve scalability issues. We focus on Janus Gateway's producer-consumer model, where plugin and core threads are decoupled, and packets are relayed via queues. This inter-thread communication incurs synchronization and memory management overhead, which significantly degrades performance. As traffic volume grows, more event-loop and helper threads are spawned; however, once the number of threads exceeds the available CPU cores, frequent context switches occur, further exacerbating performance degradation.

III. Architecture

To address the aforementioned scalability issues, this paper proposes Janus Streamer, an enhanced version of the Janus Gateway. Janus Streamer bypasses the producer-consumer model by allowing helper threads to transmit packets directly, without passing through the event-loop. It retains the core structure of the Janus Gateway while introducing a new data plane for streaming tasks, thereby eliminating inherent overheads and reducing context switches between the core and plugins.

A. Overview

Figure 3 illustrates the overall architecture of Janus Streamer. Janus Streamer extends the Janus Gateway architecture by integrating the existing framework with an additional data plane specifically designed for efficient packet transmission. The core design of Janus Streamer differs from that

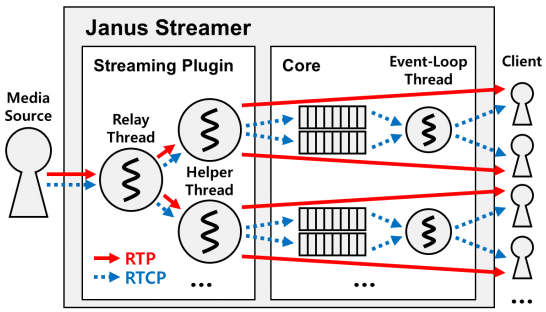


Fig. 3: Architecture of Janus Streamer

of Janus Gateway by introducing enhanced functionality that allows plugins to send packets directly to clients – a feature not natively supported by Janus Gateway. The streaming plugin of Janus Streamer leverages this capability, sending data packets directly from the plugin threads, as illustrated by the red solid arrows in Figure 3, thereby establishing a new data plane. We further optimized this new data plane by utilizing `sendmmsg`, Generic Segmentation Offload (GSO) and CPU affinity to increase transmission bandwidth. For RTCP relaying, Janus Streamer continues to use the existing queue-based event-loop, as indicated by the blue dotted arrows in Figure 3. The source code is publicly available at https://github.com/snu-csl/janus-streamer.

B. Design and Implementation

We implemented a new interface called `relay_rtps` for use by plugins, which enables direct packet transmission, bypassing the core thread. In the existing Janus Gateway, RTP and RTCP packets are transmitted through the core event-loop using `janus_ice_outgoing_traffic_handle`, which performs RTP validation, extension updates, retransmission buffering, SRTP encryption, and statistics updates. To expose this functionality to plugins, we refactored the transmission code into a new core function, `janus_ice_relay_rtps`, and connected it to the plugin system via the `relay_rtps` interface in the `janus_callbacks` structure. This allows plugin threads to send validated RTP packets directly, without relying on the core event-loop.

Direct packet transmission from the plugins effectively addresses the overhead issues of the existing Janus Gateway. Unlike the conventional Janus Gateway, which requires context switches to the core's event-loop to pop handle queues for packet transmission, Janus Streamer eliminates these steps. By enabling the streaming plugin to send packets directly from the plugin threads, the packets bypass the event-loop thread, thereby reducing context switches. Additionally, direct packet transmission skips the handle queue, reducing the overhead of queue manipulation and memory management for queued packets. Consequently, Janus Streamer offers a substantial performance improvement, especially beneficial for high-performance applications that handle large data packets.

Janus Streamer is designed to be backward-compatible with existing Janus Gateway systems by reusing the same structural framework and adding only a few new core functions. It retains the original producer–consumer model while providing an interface that allows plugins to transmit packets directly, preserving the core–plugin separation that enables extensibility without modifying the core. Plugin developers can choose to transmit data packets through the existing queue and event-loop using `relay_rtp` or bypass them using the new `relay_rtps` interface. This design simplifies system migration toward enhanced streaming functionality with Janus Streamer.

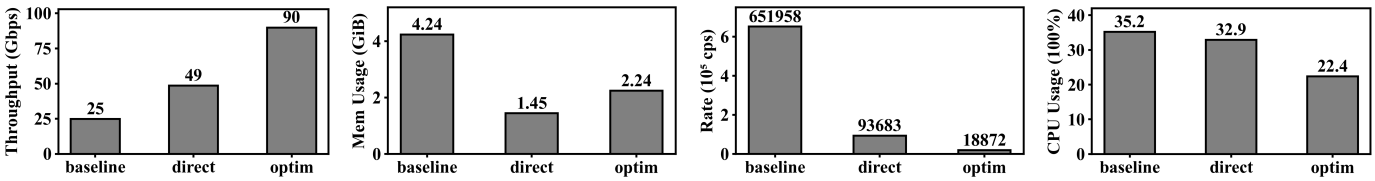
The new core functions introduced in Janus Streamer are not limited to streaming applications. The direct packet transmission capability is implemented by providing a new interface that allows plugins to use a feature previously reserved for the core. The new interface can also be utilized by other plugins within the system to achieve similar performance improvements. This versatility ensures that the Janus Streamer serves as a robust and flexible enhancement to the Janus Gateway, offering broad applicability across a range of use cases where optimized packet relaying is crucial.

C. Optimizing the data plane

Janus Streamer spends a significant portion of its CPU cycles on packet transmission, so optimizing it can lead to substantial performance improvements. Because WebRTC relies on UDP-based RTP for media delivery, we optimized the transmission of UDP packets. We utilize `sendmmsg`, Generic Segmentation Offload (GSO) and CPU affinity to enhance the performance of Janus Streamer's dedicated data plane. By implementing these techniques, we have accelerated the packet transmission, resulting in notable gains in overall system efficiency and throughput.

The `sendmmsg` system call enables the transmission of multiple messages over a socket with a single system call. It is available only on Linux and was introduced in Linux 3.0 [21]. The `sendmmsg` system call reduces the number of system calls required for packet transmission and also lowers the overhead associated with invoking packet transmission libraries. The existing Janus Gateway uses GLib [22] to transmit packets, repeatedly accessing the socket descriptor and remote address for each transmission. By using `sendmmsg`, this process can be performed only once, leading to notable performance improvements.

UDP Generic Segmentation Offload (UDP GSO) provides an effective method for applications to transmit packets larger than the Maximum Transmission Unit (MTU) size. UDP GSO splits large byte streams into gso-sized segments, copies the UDP and IP headers, and appends the computed UDP checksum field via the NIC (Network Interface Card) to generate multiple packets [23]. By reducing the overhead associated with generating multiple packets, UDP GSO decreases packet transmission latency and improves throughput. This process differentiates UDP GSO from IPv4 fragmentation, where the latter requires IP reassembly. UDP GSO transmits complete UDP packets, thereby eliminating the need for reassembly. This can be used by setting the `msg_control`



(a) Throughput (b) Memory Usage (c) Context Switches (d) CPU Usage

Fig. 4: Performance of Janus Streamer (direct, optim) vs. Janus Gateway (baseline) on 40 CPU cores for 5 Mbps streaming

field of the `msghdr` structure and invoking `sendmsg` or `sendmmsg` [23].

To leverage UDP GSO for RTP streams, we designed a packet merging algorithm that combines RTP packets into GSO-compatible datagrams. The algorithm appends packets of equal size to a base datagram, optionally including smaller ones to fill the remaining space, while deferring larger packets for future merges. After merging, the packets are concatenated into a buffer, the first datagram's length is set to `gso_size`, and both the buffer and size are passed to the socket. This method is effective in streaming because RTP packets—especially those from large video frames—often have identical lengths.

Finally, we preserved the CPU core affinity for the streaming plugin's helper threads. Previous studies [24], [25] have shown that fixing CPU core affinity in the data plane improves performance. All of our optimization methods were designed to ensure that a single helper thread can handle packet transmission independently. By binding the thread to a specific CPU core, we avoided cache-line contention, which is crucial for reducing context switching overhead.

IV. Performance evaluation

A. Methodology

We configured our test environment using a single-server setup to stream both video and audio. Janus Streamer was developed based on Janus Gateway v1.2.3 and deployed on a 40-core Intel Xeon Gold 5218R server running Linux kernel version 5.4.0. The server was equipped with a Mellanox ConnectX-5 100GbE NIC. Following best practices recommended by Netflix [26] and Google [27], we used RTP streams generated via GStreamer, consisting of VP8-encoded video at bitrates of 5, 10, 20, 40, and 80 Mbps, alongside Opus-encoded audio at 0.2 Mbps.

We conducted the experiment by connecting load-generating clients and a QoE-monitoring client to Janus Streamer. Both client types were implemented using Pion packages [28]. Media transmission was initiated once all connections were successfully established. On the server side, we measured transmission bitrate, CPU usage, and context-switching rate. On the client side, we measured receive bitrate, NACK rate, packet loss rate, jitter, and VMAF score – a perceptual video quality assessment algorithm developed by Netflix [29], [30]. Metrics were collected every 5 seconds over a 10-minute streaming period, and averages were computed for analysis. In the baseline experimental setup, Janus Gateway achieved a VMAF score of 99, an audio jitter of 20 ms, and a packet loss rate of 0% for a single client.

To evaluate the performance of the streaming systems, we measured the maximum number of clients that could sustain the same level of QoE metrics as the baseline client. The number of concurrent clients was incrementally increased in steps corresponding to approximately 200 Mbps of aggregate bitrate, until the QoE thresholds were no longer satisfied. For example, each step added 40 clients for 5 Mbps streams and 2 clients for 80 Mbps streams. QoE was assessed using three criteria: a VMAF score above 93 [31], audio jitter below 40 ms, and a packet loss rate under 1%.

B. Performance with all CPU cores

In an environment with 40 CPU cores, we confirmed that Janus Streamer achieves the maximum throughput supported by the NIC. Figure 4 compares Janus Streamer under direct packet transmission (direct) and full optimizations (optim) against Janus Gateway (baseline). As shown in Figure 4a, Janus Streamer shows a 2x improvement in throughput by employing direct packet transmission, reaching 90 Gbps with all optimizations applied – the maximum bandwidth measured by `iperf` in our experimental setup.

These optimizations significantly reduce context switches, memory usage, and CPU usage. In Janus Gateway, the separation of core and plugin threads often results in frequent context switches and additional memory allocation as packets pass through queues, particularly when the thread count increases to support more connections. In contrast, as shown in Figure 4b and 4c, Janus Streamer reduces memory usage by 48% and context switches by 98%. Additionally, it demonstrates improved CPU efficiency, with a 37% reduction in CPU usage while supporting a large number of clients, as illustrated in Figure 4d.

C. Performance on a single CPU core

Janus Streamer's data plane demonstrates efficient utilization of a single CPU core. Figure 5 shows the maximum throughput at each optimization stage when streaming five different source bitrates, using a single static event-loop thread and a single helper thread. Beyond the baseline, direct packet transmission by the helper thread allows for an accurate assessment of maximum throughput on a single core. The results indicate a consistent throughput increase with each optimization. Notably, the introduction of GSO optimization

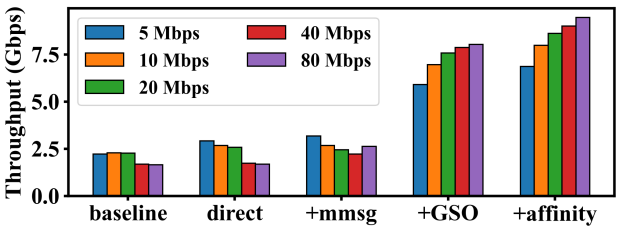


Fig. 5: Throughput at each optimization stage with 1 static event-loop and 1 helper thread for 5 Mbps to 80 Mbps streams

significantly boosts performance at higher bitrates (e.g., 80 Mbps) by merging larger video frames into larger packets. Comparing the final optimization stage (affinity) with the baseline, the throughput improves by 3.1x, 3.5x, 3.8x, 5.3x, and 5.7x for the five streaming sources. These substantial performance gains are particularly impressive given that they are achieved using the Linux kernel.

V. Conclusion

In this work, we introduced Janus Streamer, a drop-in replacement for Janus Gateway, engineered to scale effectively in high-load streaming environments. Janus Streamer enhances Janus Gateway's modular architecture by incorporating a direct packet transmission interface while maintaining backward compatibility. To support this, it incorporates a data plane optimized with `sendmmsg`, Generic Segmentation Offload (GSO), and CPU affinity.

Janus Streamer offers an effective and easily applicable optimization method for media transmission servers, particularly for high-bitrate workloads with uniformly sized RTP packets. Our results show a 3.6x increase in throughput and a 37% reduction in CPU usage when using 40 CPU cores. On a single CPU core, Janus Streamer achieves throughput improvements ranging from 3.1x at 5 Mbps to 5.7x at 80 Mbps compared to the baseline. Since all optimizations are implemented on top of the Linux kernel, they are expected to be adaptable to a wide range of media transmission systems.

VI. Acknowledgements

This work was supported by the Institute of Information & Communications Technology Planning & Evaluation (IITP) grant (No. RS-2021-0-01363, RS-2024-00349594), funded by the Korean government (MSIT).

References

[1] H. Alvestrand, "Rfc 8825: Overview: Real-time protocols for browser-based applications," Jan 2021.
[2] S. Turner and T. Hardie, "Real-time communication in web-browsers (rtcweb)," 2024.
[3] C. Jennings, F. Castelli, H. Boström, and J.-I. Bruaroey, "Webrtc: Real-time communication in browsers," Mar 2023. [Online]. Available: https://www.w3.org/TR/2023/REC-webrtc-20230306/
[4] Meetecho, "Meetecho/janus-gateway: Janus webrtc server," Apr 2024. [Online]. Available: https://github.com/meetecho/janus-gateway
[5] A. Amirante, T. Castaldi, L. Miniero, and S. P. Romano, "Janus: a general purpose webrtc gateway," in *Proceedings of the Conference on Principles, Systems and Applications of IP Telecommunications*, 2014, pp. 1–8.
[6] A. Elangovan, "Build a webrtc video calling app with janus server," Sep 2024. [Online]. Available: https://www.mirrorfly.com/blog/janus-webrtc-media-server-video-calling-app/
[7] G. P. Inquiries, Feb 2022. [Online]. Available: https://news.dolby.com/en-WW/209521-dolby-acquires-millicast-the-real-time-ultra-low-delay-video-streaming-platform
[8] A. Media, "Ultra low latency webrtc live streaming media server," Nov 2024. [Online]. Available: https://antmedia.io/
[9] LiveSwitch, "Why liveswitch server," Nov 2024. [Online]. Available: https://developer.liveswitch.io/liveswitch-server/index.html
[10] B. Garcia, L. Lopez-Fernandez, M. Gallego, and F. Gortazar, "Kurento: the swiss army knife of webrtc media servers," *IEEE Communications Standards Magazine*, vol. 1, no. 2, pp. 44–51, 2017.
[11] Jitsi, "Jitsi meet - secure, simple and scalable video conferences that you use as a standalone app or embed in your web application," Nov 2024. [Online]. Available: https://github.com/jitsi/jitsi-meet
[12] Meedoze, "Medooze: A future proof, experimental webrtc vp9 svc sfu wit end to end encryption support," Nov 2024. [Online]. Available: https://github.com/medooze/sfu
[13] Mediasoup, "Mediasoup: Cutting edge webrtc video conferencing," Nov 2024. [Online]. Available: https://github.com/versatica/mediasoup
[14] E. André, N. Le Breton, A. Lemesle, L. Roux, and A. Gouaillard, "Comparative study of webrtc open source sfus for video conferencing," in *2018 Principles, Systems and Applications of IP Telecommunications (IPTComm)*. IEEE, 2018, pp. 1–8.
[15] Á. Martín, D. Mejías, Z. Fernández, R. Viola, J. Pérez, M. García, G. Velez, J. Montalbán, and P. Angueira, "Adaptive qos of webrtc for vehicular media communications," in *2022 IEEE International Symposium on Broadband Multimedia Systems and Broadcasting (BMSB)*. IEEE, 2022, pp. 1–6.
[16] F. Kilic, M. Hassan, and W. Hardt, "Prototype for multi-uav monitoring–control system using webrtc," *Drones*, vol. 8, no. 10, p. 551, 2024.
[17] J. Cao, K. Y. Lam, and L.-H. Lee, "Real-time webxr edge-based object detection for ar," in *Proceedings of the 21st Annual International Conference on Mobile Systems, Applications and Services*, 2023, pp. 590–591.
[18] H. Schulzrinne, S. Casner, R. Frederick, and V. Jacobson, "Rfc3550: Rtp: A transport protocol for real-time applications," 2003.
[19] C. Perkins, M. Westerlund, and J. Ott, "Rfc 8834: Media transport and use of rtp in webrtc," Jan 2021.
[20] E. Carrara, K. Norrman, D. McGrew, M. Naslund, and M. Baugher, "Rfc 3711: The secure real-time transport protocol (srtp)," Mar 2004.
[21] F. S. Foundation, "sendmmsg(2) manual page," Jun 2024. [Online]. Available: https://man7.org/linux/man-pages/man2/sendmmsg.2.html
[22] G. D. Team, "Glib – 2.0," 2025. [Online]. Available: https://docs.gtk.org/glib/
[23] W. De Bruijn and E. Dumazet, "Optimizing udp for content delivery: Gso, pacing and zerocopy," in *Linux Plumbers Conference*, 2018.
[24] Z. Li, "Hpsrouter: A high performance software router based on dpdk," in *2018 20th International Conference on Advanced Communication Technology (ICACT)*. IEEE, 2018, pp. 503–506.
[25] E. Jeong, S. Wood, M. Jamshed, H. Jeong, S. Ihm, D. Han, and K. Park, "{mTCP}: a highly scalable user-level {TCP} stack for multicore systems," in *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, 2014, pp. 489–502.
[26] Netflix, "Internet connection speed recommendations," 2024. [Online]. Available: https://help.netflix.com/en/node/306
[27] Google, "Youtube recommended upload encoding settings," 2024. [Online]. Available: https://support.google.com/youtube/answer/1722171
[28] Pion, "Pion/webrtc: Pure go implementation of the webrtc api," 2024. [Online]. Available: https://github.com/pion/webrtc
[29] B. García, L. López-Fernández, F. Gortázar, and M. Gallego, "Practical evaluation of vmaf perceptual video quality for webrtc applications," *Electronics*, vol. 8, no. 8, p. 854, 2019.
[30] Netflix, "Netflix/vmaf: Perceptual video quality assessment based on multi-method fusion." 2023. [Online]. Available: https://github.com/Netflix/vmaf
[31] R. Rassool, "Vmaf reproducibility: Validating a perceptual practical video quality metric," in *2017 IEEE international symposium on broadband multimedia systems and broadcasting (BMSB)*. IEEE, 2017, pp. 1–2.