

A WebRTC-Based Architecture for Remote Monitoring of Mobile Robots

Gabriel Víctor M. Muniz¹, Michel Melo da Silva¹ and Alexandre Santos Brandão¹

Abstract—Remote monitoring and teleoperation of robotic systems over the Internet require low-latency and bandwidth-efficient video transmission to ensure safe and responsive control. This article presents the development and experimental validation of a real-time video streaming architecture based on the WebRTC protocol, integrated into the ROS ecosystem using GStreamer. The proposed solution exploits hardware-accelerated video encoding on an NVIDIA Jetson Nano to optimize the transmission of RGB-D camera data over public networks. Connectivity across distinct networks and restrictive NAT and firewall environments is ensured through a TURN-assisted signaling mechanism. Experimental results evaluate glass-to-glass latency and bandwidth consumption, comparing the performance of H.264 and VP8 codecs at multiple video resolutions. The results demonstrate that hardware-accelerated H.264 significantly reduces bandwidth usage while maintaining low and stable latency suitable for real-time robotic teleoperation, whereas VP8 is only viable at low resolutions. Overall, the proposed architecture provides a feasible solution for remote robotic monitoring and control over public networks.

I. INTRODUCTION

In recent decades, intelligent robotics has taken on a central role in technological advancement, driving the transition to Industry 4.0. Modern applications require robots to operate not only autonomously, but also collaboratively and remotely supervised in environments ranging from production lines, hard-to-reach locations, and even disaster zones [1]. To enable this operation, real-time monitoring through visual sensors is essential. However, modern depth cameras generate a massive volume of data, often requiring transfer rates exceeding 25 MB/s for raw streams, creating significant transmission bottlenecks when there is no efficient compression strategy [2]. The problem worsens when the operation exceeds the limits of local area networks (LAN). The standard robotics middleware, the Robot Operating System (ROS), uses serial messaging mechanisms that, although effective locally, become inefficient for transmitting raw video over the internet, consuming bandwidth and introducing excessive latencies [2]. Traditional transmission solutions, such as the RTMP protocol, although robust, often have latencies exceeding three seconds, which makes secure real-time teleoperation unfeasible [3]. Alternatively, cloud-based approaches solve connectivity issues but add operational costs and dependence on external infrastructure [4].

*This work was supported by FAPEMIG and CNPq.

¹The authors are with the Núcleo de Especialização em Robótica, Department of Electrical Engineering, and Graduate Program on Computer Science, Federal University of Viçosa, Viçosa - MG, 36570-009, Brazil, E-mail: {gabriel.muniz, michel.m.silva, alexandre.brandao}@ufv.br

In this context, this paper proposes a video transmission architecture that integrates the Web Real-Time Communication (WebRTC) protocol into the ROS ecosystem, seeking to mitigate the limitations of the current infrastructure. The approach seeks to establish direct Peer-to-Peer (P2P) communication between the robot and the control station. The main purpose is to ensure the stability and fluidity of teleoperation, guaranteeing the delivery of visual data even in scenarios of restricted connectivity, where traditional methods fail due to excessive bandwidth consumption [4].

To validate the effectiveness of the proposed solution, the study conducts a comparative performance analysis, contrasting the efficiency of H.264 and VP8 codecs operating at multiple video resolutions. The evaluation focuses on metrics critical to real-time systems, such as glass-to-glass latency, frame rate, and resource consumption. In this way, we seek to quantify the operational gains of inter-frame compression, demonstrating the feasibility of a scalable architecture for connected mobile robotics [2].

II. RELATED WORKS

A. WebRTC

WebRTC is a technology that enables real-time P2P communication, initially designed to enable audio and video calls, chats, and data sharing in browsers without the need for external plugins [5]. Through this protocol, it is possible to transmit video, audio, and also arbitrary data through data channels. In addition, WebRTC integrates several fundamental technologies, such as media codecs, compression, automatic gain control, retransmission mechanisms, and advanced negotiation and transport protocols (SDP, ICE, DTLS, and SRTP) [4]. These features make WebRTC an efficient option for robotic applications that require low latency, adaptability to variable networks, and reliable P2P communication, surpassing traditional protocols such as RTMP in teleoperation scenarios [1], [3].

B. GStreamer

GStreamer is a widely used multimedia framework for real-time audio and video processing, streaming, and manipulation [6]. Its architecture is based on pipelines, which represent graphs composed of interconnected elements. Each element performs a specific function: capture, encoding, decoding, multiplexing, analysis, or transmission. In the context of cloud robotics and distributed systems, the use of GStreamer pipelines enables the application of efficient encoding (such as H.264 and VP8), which is essential for reducing bandwidth consumption in wireless networks [2].

C. ROS

ROS is a modular, open-source middleware widely used in the development of robotic systems [7]. Its goal is to facilitate code reuse and integration between sensors, actuators, and algorithms through a distributed architecture based on the publish/subscribe model. Due to its widespread adoption in industry, research, and education, ROS has become fundamental to applications such as mobile robots, manipulators, and cooperative systems [1]. However, its message transport architecture presents performance challenges when dealing with large uncompressed data streams, motivating integration with dedicated streaming tools [2].

ros-gst-bridge is a community library that establishes a bidirectional bridge between ROS and GStreamer [8]. It provides GStreamer elements capable of publishing or receiving ROS topic messages directly into a multimedia pipeline. This way, data such as images, sensor messages, or audio streams can be automatically converted between the ROS message format and GStreamer buffers, facilitating hardware encoding and transport via WebRTC.

III. THE SYSTEM OVERVIEW

The architecture developed in this work reproduces a real scenario of robotic teleoperation via the internet, where the operator and the robot are on different networks, requiring mechanisms for NAT (Network Address Translation) transposition. The system consists of four main entities: the robot, the signaling server, the TURN (Traversal Using Relays around NAT) server, and the remote control station.

The Intel RealSense D435 RGB-D camera was used for image acquisition. This camera is capable of providing video streams at various resolutions and frame rates. These characteristics allow different workloads to be simulated in transmission tests, representing the bandwidth obstacles that this work aims to solve.

The NVIDIA Jetson Nano was adopted as the embedded computer. Unlike generic single-board computers, the Jetson Nano has an integrated GPU based on the Maxwell architecture. This feature allows for the use of hardware-accelerated encoders (NVENC), enabling the H.264 compression necessary to transmit fluid video over the internet, freeing up the main CPU for navigation and control tasks.

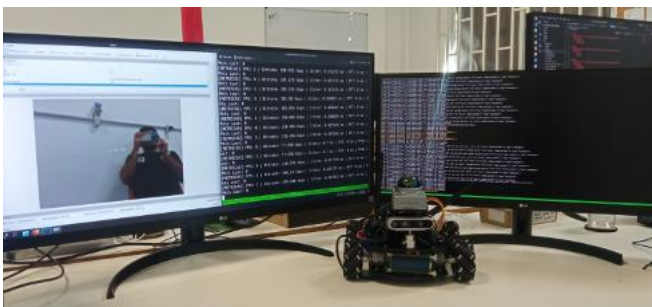


Fig. 1. Experimental LAN test setup. The left monitor displays the robot camera stream and performance metrics, while the right monitor executes the onboard control and processing code.

The network infrastructure was configured to operate in long-distance scenarios using a Wide Area Network (WAN). Since the robot and control station are located on separate networks, direct P2P connections are often blocked by firewalls and restrictive NATs, a common limitation in institutional networks such as the institutional Wi-Fi network used in this study. To overcome this obstacle, a TURN server provided by the OpenRelay service [9] was integrated into the architecture. This component acts as an intermediary, retransmitting media packets whenever direct negotiation fails, ensuring the controllability of the robot regardless of network restrictions [4].

The control station was implemented on a computer external to the robot network. This unit runs the GStreamer receiver client, negotiating the connection with the help of the signaling server and TURN, decoding the received stream, and republishing it on the local ROS network for viewing. The hardware topology and data flow over the internet are illustrated in Figure 2.

IV. METHODS AND SYSTEM IMPLEMENTATION

This section details the implementation of the transmission system, whose architecture prioritizes modularity and real-time execution efficiency. The system was developed in C++ using the ROS 2 framework, encapsulating the transmission functionalities in dedicated nodes that operate asynchronously. Multimedia integration is performed through the *ros-gst-bridge* library, which allows GStreamer pipelines to be instantiated directly within the ROS node lifecycle.

The system consists of three main software modules: the Transmitter node, the Receiver node, and the Signaling Server (both on the receiving computer). Figure 3 illustrates the connection diagram. All code used in this article is available on GitHub [10].

A. Transmitter node

The transmitter node is responsible for ingesting, encoding, and sending the video stream. Its implementation uses the “*rosimagesrc*” element to subscribe to the raw image topic (`/camera/camera/color/image_raw`), provided by Intel’s official wrapper for ROS 2 [11]. The node converts the received ROS messages to the I420 (YUV) color space, the native format required by video encoders.

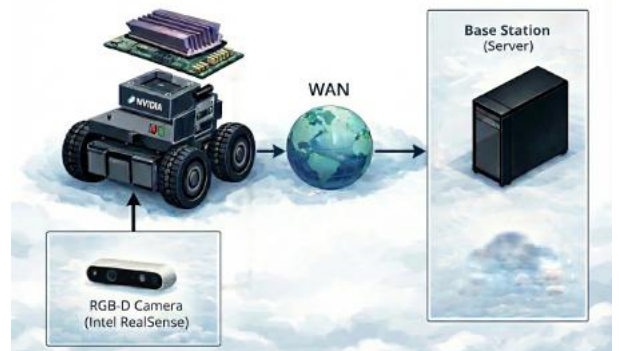


Fig. 2. System hardware architecture.

The GStreamer pipeline structure was configured with a focus on maintaining a balance between performance and quality, adopting two main strategies. First, in the VP8 scenario, the “vp8enc” element was configured with the parameters “deadline=1” and “threads=2.” These adjustments instruct the encoder to prioritize processing speed over maximum compression efficiency, an essential requirement for real-time operations, forcing the use of Jetson Nano’s processing resources. For the scenario using H.264, the “nv412h264enc” element was configured with “bitrate=2000000,” “insert-sps-pps=true,” and “maxperf-enable=1.” These settings seek to ensure network transmission stability, compatibility with remote decoders, and minimization of hardware encoding latency. Secondly, restrictive buffer management was implemented using “queue” elements with the “leaky=downstream” property and limited “max-size-time”. This configuration ensures that, in cases of network congestion or processing bottlenecks, older frames are discarded immediately, ensuring that the remote operator always sees the most recent status of the robot.

The encoded stream is delivered to the “webrtcbin” element, which manages the session. The code implements NAT traversal logic, adding STUN and TURN servers provided by the Metered service [12], ensuring connectivity even through restrictive firewalls.

B. Receiver node

The receiver node operates at the control station and performs the reverse process. It implements a WebSocket client via the ixwebsocket library to negotiate the session with the transmitter. A critical aspect of the receiver implementation is the handling of dynamic pads. Since the WebRTC stream is negotiated at runtime, the “webrtcbin” element does not initially have any connected video outputs. The software addresses this feature through a callback that communicates directly with “webrtcbin.” When a new media stream is detected, the code checks for format compatibility, either VP8 or H.264, and dynamically connects to the rest of the decoding pipeline.

Finally, the video is converted to BGR format and delivered to the “rosimagesink” element, which publishes it to a separate receiver (/received_camera/image_raw) topic, making it accessible for viewing in RViz or processing by other ROS nodes.

C. Signaling Mechanism

To enable the initial exchange of metadata (SDP and ICE candidates), a signaling server was implemented in Python, using the websockets library on the asynchronous asyncio framework. The server acts as an agnostic JSON message router, not interpreting the content of the WebRTC negotiation, but only forwarding it.

The established communication protocol follows a URL identification logic, where each node connects to the server by informing its unique identifier in the request path (e.g., ws://address:port/CLIENT.ID), allowing the server to maintain a memory mapping of active connections.

Routing operates sequentially: the client sends a JSON object containing the “id” field with the desired destination. The server extracts this destination and locates the corresponding socket in the table of active clients. Before final forwarding, the server injects the sender’s ID into the message body. This strategy ensures that the recipient identifies the source of the ICE offer or candidate without ambiguity.

To ensure public accessibility to the signaling server during the experiments, the ngrok tool [13] was used, which establishes a secure route, exposing the local server port to the internet and bypassing NAT restrictions on the network where the server was hosted.

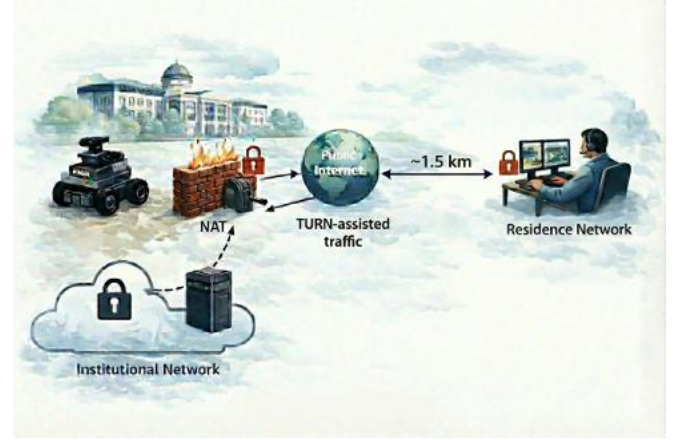


Fig. 3. Network connection diagram.

D. Experiment Setup

The transmitting node was connected to the university network, which is located behind institutional firewalls and restrictive NAT. The receiving node was positioned in a residence located approximately 1.5 km away, connected to the internet through a different internet service provider. This topology forces data traffic to travel over the public internet, using TURN servers for NAT traversal. This subjects the system to routes that are more consistent with reality, with variations in latency and bandwidth.

E. Synchronization and Latency Measurement

The critical metric in this study is glass-to-glass latency, defined as the time interval between the frame captured by the camera and its display on the operator’s screen. To measure this quantity, it was necessary to ensure synchronization between the transmitter and receiver clocks. For this purpose, the NTP (Network Time Protocol) was used through the chrony service, which adjusts the system clocks, minimizing the time difference between different machines.

The measurement method consists of timestamping at the moment the frame is generated. The transmitting node captures the current system time ($\mathcal{T}_{tx}$) and appends this information to the video frame being sent. Upon receiving and decoding the frame, the receiving node records the arrival time ($\mathcal{T}_{rx}$). The latency ($\mathcal{L}$) is then calculated by the

difference:

$$\mathcal{L} = \mathcal{T}_{rx} - \mathcal{T}_{tx} \quad (1)$$

Since both devices are synchronized via chrony, this difference reflects the total processing and transmission delay, but does not guaranty absolute accuracy.

F. Network Metrics Collection

In addition to latency, essential quality parameters for video performance analysis were monitored. Bitrate, packet delay variation (jitter), and frames per second (FPS) metrics were programmatically extracted from internal statistics provided by GStreamer’s own webrtcbin element. Access to these statistics allowed us to monitor connection stability and the response of codecs to network fluctuations during testing.

V. RESULTS AND DISCUSSION

The experiments conducted aimed to quantify the impact of hardware acceleration on glass-to-glass latency and video stream stability in a robotic teleoperation system. The tests were conducted in a WAN scenario, varying between resolutions of 1080p at 6 FPS and 720p and 480p at 30 FPS, comparing the performance of the H.264 codec via NVENC hardware against VP8 via software on the CPU. It should be noted that this comparison is inherently asymmetric. H.264 encoding was performed using the hardware-accelerated encoder available on the Jetson Nano, while VP8 encoding relied exclusively on the CPU, as the platform does not provide a hardware-accelerated VP8 encoder. Consequently, the observed performance degradation primarily reflects the computational limitations of CPU-based encoding rather than an intrinsic inefficiency of the VP8 codec itself.

A. H.264 Codec Performance

Performance was evaluated using the “nvv4l2h264enc” element, which exploits the dedicated hardware of the Jetson Nano. The results, summarized in Table I, demonstrated high processing efficiency, with a direct impact on transmission consistency. The 1080p configuration exhibited a higher average latency. However, further analysis indicates that this delay is not caused by the encoding process itself, but rather by the camera capture interval. At a rate of 6 FPS, a new frame is generated approximately every 166 ms, introducing a delay at the source even before the encoding stage begins.

TABLE I

H.264 GLASS-TO-GLASS LATENCY STATISTICS VIA NVV4L2H264ENC

Resolution	FPS	Average (μ)	Deviation (σ)
1920x1080	6	241 ms	± 108.3 ms
1280x720	30	182 ms	± 45.8 ms
640x480	30	169 ms	± 25.5 ms

TABLE II

VP8 GLASS-TO-GLASS LATENCY STATISTICS VIA VP8ENC

Resolution	FPS	Average (μ)	Deviation (σ)
1920x1080	6	461.4 ms	± 215.0 ms
1280x720	30	260.0 ms	± 144.6 ms
640x480	30	189.9 ms	± 33.3 ms

By increasing the frame rate to 30 FPS and adjusting the resolution to 720p and 480p, latency dropped dramatically, stabilizing below 190 ms. The 640x480 resolution not only delivered the lowest average latency, but also had a standard deviation of only 25.5 ms.

B. VP8 Codec Performance

Performance was evaluated using the “vp8enc” encoder, which relies entirely on CPU-based compression. This configuration highlights the processing bottleneck of the system. As shown in Table II, a significant performance degradation is observed, with clear evidence of CPU saturation. In the 1080p scenario, the average latency nearly doubled compared to H.264, with the occurrence of extreme peaks, rendering teleoperation impractical.

Furthermore, visual inspection revealed that, under high CPU load, key frames were not generated at the correct intervals. As a result, subsequent predictive frames (P-frames) were incorrectly reconstructed, leading to severe visual artifacts that persisted for several seconds.

Another relevant observation was the drop in frame rate. Even when configured for 30 FPS, the VP8 encoder at 720p achieved an average of only about 15 FPS, highlighting the CPU inability to sustain the required encoding rate. This limitation led to the formation of buffer queues, which in turn increased both latency and jitter.

C. Transmission Quality Analysis

In addition to latency, the viability of remote monitoring and teleoperation depends on the health of the data flow. Figure 4 presents a visual comparison of the collected network metrics, highlighting the bandwidth efficiency and delivery stability between the two codecs.

The analysis of the results shows a clear distinction between internal processing limitations and external network interference. While the VP8 codec suffered degradation caused by CPU bottlenecks, collapsing at higher resolutions, H.264 maintained robust stability, with the fluctuations observed in the 480p test resulting only from momentary instabilities in the communication channel, not the hardware.

In addition to its processing inefficiency, VP8 proved to be significantly more bandwidth-intensive, requiring up to five times more data than H.264 at 1080p while delivering a lower-quality and less stable video stream.

The correlation between frame rate instability and the high standard deviation observed in VP8 highlights video stream instability. This translates into an oscillation in the frame arrival interval, turning delay into an unpredictable variable rather than a constant. In real-time control systems, this variability is more harmful than absolute latency, as it introduces uncertainty into the visual feedback received by the operator. On the other hand, the metric consistency of H.264 suggests that hardware acceleration was effective in maintaining transmission predictability, ensuring that frames were delivered at regular intervals, a mandatory feature for maintaining teleoperation reliability.

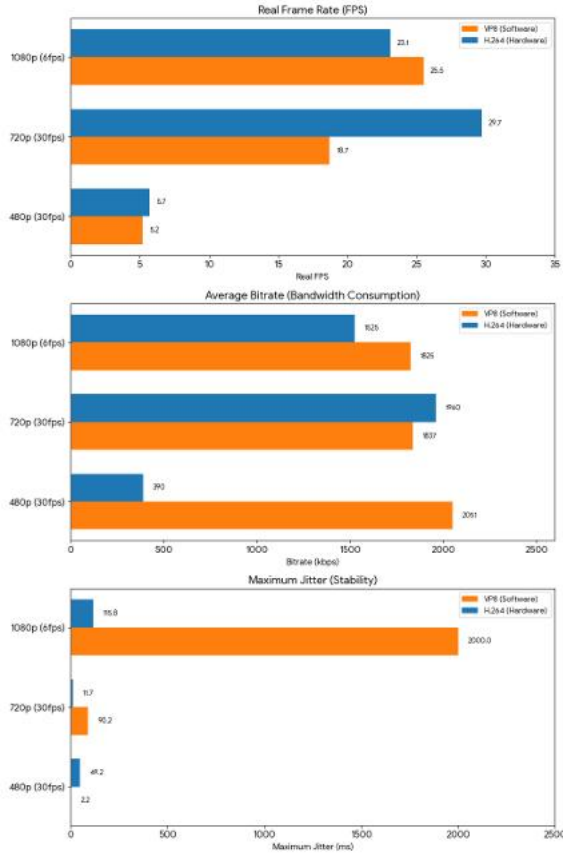


Fig. 4. Visual comparison of actual frame rate, average bitrate, and maximum jitter between H.264 and VP8 codecs across tested resolutions.

The comparative analysis presented in Table III consolidates the experimental findings by translating quantitative performance metrics into practical implications for real robotic applications. The results indicate that hardware-accelerated H.264 consistently provides low and predictable latency, reduced bandwidth consumption, and stable frame delivery, which are essential requirements for real time monitoring and teleoperation systems operating over public

networks. In contrast, VP8 exhibits severe limitations on resource constrained embedded platforms, with high latency variability, frame drops, and visual artifacts that compromise operational reliability, particularly in human in the loop control scenarios. Therefore, while VP8 may be acceptable for low resolution and non critical monitoring tasks, the experimental evidence demonstrates that H.264 is the most suitable choice for ground robots, aerial platforms, and remote teleoperation applications that demand responsiveness and predictable visual feedback.

VI. CONCLUDING REMARKS

This work presented an architecture for the teleoperation of mobile robots via the internet, overcoming the bandwidth and connectivity limitations of native ROS. The integration of WebRTC with NVIDIA Jetson Nano hardware acceleration enables real-time video transmission with low latency, even across separate networks protected by NAT.

The experiments confirm that processing on the Jetson Nano is the primary bottleneck when using the VP8 encoder. This finding highlights the impracticality of employing this codec for HD resolutions, as it saturates the CPU and leads to system instability. In contrast, the use of the hardware-based H.264 encoder offloads the processor and ensures the level of fluidity required for control, even at higher resolutions.

The results indicate that the optimal operating configuration is achieved with H.264 at a resolution of 640×480, providing consistent latencies below 170 ms. This setup enables reliable remote monitoring and offers sufficient responsiveness for teleoperation tasks.

For future work, we suggest using video stream for dynamic robot control and implementing WebRTC data channels to unify the transmission of control commands and decision data. In addition, extending the proposed architecture to support the transmission of point clouds generated by stereo cameras and 3D LiDAR sensors would enable a more comprehensive evaluation of network and processing limits, particularly under increased computational demand and higher bandwidth requirements typical of perception tasks.

TABLE III

COMPARISON BETWEEN H.264 AND VP8 VIDEO TRANSMISSION APPROACHES FOR PRACTICAL ROBOTIC APPLICATIONS

Application Scenario	H.264 (Hardware-Accelerated)	VP8 (Software-Based)
Ground robot short-range monitoring	Suitable. Provides stable low latency and reduced bandwidth consumption, enabling continuous visual feedback even on constrained networks.	Suitable only at low resolutions. Performance degrades significantly with increased resolution due to CPU overload.
UAV monitoring and inspection	Suitable. Low and predictable latency supports safe supervision and real time decision making when combined with hardware encoding.	Not suitable. High processing times and FPS instability would cause periods of vision loss, breaking real time monitoring.
Remote monitoring over public networks	Suitable. Good stability and low latency even at 1080p, as few frames would already be enough in this case.	Suitable, though jitter and frame drops reduce monitoring reliability under variable network conditions, causing some blind time.
Teleoperation with human in the loop	Suitable. Stable frame rate and low jitter provide predictable visual feedback, which is critical for safe and responsive teleoperation.	Not suitable. High latency variability and visual artifacts introduce uncertainty, negatively impacting operator performance.
Resource constrained embedded platforms	Efficient. Offloads encoding to dedicated hardware, preserving CPU resources for control and navigation tasks.	Inefficient. CPU saturation degrades overall system performance and control stability.
High resolution streaming (720p and above)	Supported. Maintains acceptable latency and bandwidth usage due to hardware acceleration and inter frame compression.	Not viable. Causes excessive latency, frame drops, and persistent visual artifacts.

ACKNOWLEDGMENT

The authors would like to thank FAPEMIG (Fundação de Amparo à Pesquisa do Estado de Minas Gerais) for the financial support that made this project possible, under grant numbers APQ-02282-24 and APQ-06109-24. Prof. Dr. Brandão also acknowledges continued research support from CNPq under grant no. 309884/2022-5. Mr. Muniz thanks FAPEMIG for the scholarship support.

REFERENCES

- [1] X. Wang, Y. Wu, J. Yu, Y. Wang, E. Liu, D. Zhao, and G. Liu, "Telerobot: Design and implementation of a live remote interaction platform for robots," in *2023 IEEE International Conference on Systems, Man, and Cybernetics (SMC)*. IEEE, 2023, pp. 408–412.
- [2] M. Balogh and A. Vidács, "Optimizing camera stream transport in cloud-based industrial robotic systems," *Infocommunications Journal*, vol. 14, no. 1, pp. 36–42, 2022.
- [3] T. Koyama and Y. Gotoh, "Multi-camera live video streaming over wireless network," in *Int. Conf. on Advances in Mobile Computing and Multimedia Intelligence*. Springer, 2023, pp. 144–158.
- [4] B. Anand, H. R. Kambhampaty, and P. Rajalakshmi, "A novel real-time lidar data streaming framework," *IEEE Sensors Journal*, vol. 22, no. 23, pp. 23 476–23 485, 2022.
- [5] Google WebRTC Team, "Real-time communication for the web," [Online]. Available: <https://webrtc.org>, 2024, acessado em: 06 dez. 2025.
- [6] The GStreamer Team, "Gstreamer: Open source multimedia framework," [Online]. Available: <https://gstreamer.freedesktop.org>, 2024, acessado em: 06 dez. 2025.
- [7] Open Robotics, "Ros: Robot operating system," [Online]. Available: <https://www.ros.org>, 2024, acessado em: 06 dez. 2025.
- [8] BrettRD, "ros-gst-bridge: A bidirectional bridge between ros and gstreamer," [Online]. Available: <https://github.com/BrettRD/ros-gst-bridge>, 2024, acessado em: 06 dez. 2025.
- [9] Metered, "Openrelay: Free turn server project," [Online]. Available: <https://www.metered.ca/tools/openrelay>, 2024, acessado em: 06 dez. 2025.
- [10] gabrielt17, "Ros2_webrtc: branch main-h.264," 2026, acessado em: 10 mar. 2026. [Online]. Available: <https://github.com/gabrielt17/ROS2-WebRTC/tree/main-h.264>
- [11] Intel Corporation, "Intel realsense ros wrapper," [Online]. Available: <https://github.com/realsenseai/realsense-ros>, 2024, acessado em: 07 dez. 2025.
- [12] Metered Inc., "Openrelay: Free turn server project," [Online]. Available: <https://www.metered.ca/tools/openrelay/>, 2024, acessado em: 07 dez. 2025.
- [13] ngrok, Inc., "ngrok: Unified ingress platform for developers," [Online]. Available: <https://ngrok.com>, 2024, acessado em: 07 dez. 2025.