Seeing Through the Robot's Eyes: Adaptive Point Cloud Streaming for Immersive Teleoperation

Nunzio Barone, Walter Brescia, Gabriele Santangelo, Antonio Pio Maggio, Ivan Cisternino, Luca De Cicco, Saverio Mascolo

Dipartimento di Ingegneria Elettrica e dell'Informazione
Politecnico di Bari
Via Orabona n.4, Bari, ITALY
`{nunzio.barone,walter.brescia,g.santangelo, a.maggio4,i.cisternino,luca.decicco,saverio.mascolo}@poliba.it`

Abstract. Autonomous Mobile Robots (AMRs) are increasingly deployed in diverse scenarios to automate tedious and hazardous tasks. Nonetheless, challenges such as complex environments, sensor occlusions, and limitations of autonomous navigation systems often require human intervention. Teleoperation offers a viable solution, allowing operators to remotely control AMRs when autonomy fails, without requiring physical presence. A key requirement for effective teleoperation is the real-time delivery of rich sensory information to the operator. Head-Mounted Displays (HMDs), combined with volumetric videos, provide an immersive visualization of the robot's surroundings, enabling natural viewpoint changes and improved spatial awareness compared to traditional 2D video streams. In this paper, we present a teleoperation framework to stream in real-time volumetric videos in the form of point clouds to an operator wearing a HMD. The system includes a distance-based sampling strategy that dynamically adapts the point cloud bitrate to the estimated time-varying network bandwidth, addressing constraints imposed by limited computational resources on both the robot and the HMD. The framework is implemented on a real mobile robot and evaluated under various network conditions, including a 5G connection, demonstrating its effectiveness and robustness in supporting immersive remote teleoperation. Code is available at our GitHub repository[1].

Keywords: Point cloud real-time streaming; Immersive teleoperation; mobile robots

1 Introduction

Autonomous Mobile Robots (AMRs) are increasingly adopted across several domains due to their ability to perform tasks without direct human intervention. AMRs use a range of sensors – such as LiDARs, RGB cameras, and stereo cameras – to perceive their surroundings and build environment models to support autonomous decisions and task execution.

[1] https://github.com/Diane-Spirit

Despite these capabilities, AMRs may fail to complete tasks autonomously in scenarios involving sensor occlusions, complex or cluttered environments, narrow passages, or hard-to-detect obstacles. In such cases, human intervention is required to resolve issues beyond the robot's autonomous capabilities.

In this context, teleoperation is a viable solution for remotely assisting AMRs when autonomy is compromised. It allows human operators to intervene in real-time, navigating the robot past obstacles or correcting localization and planning errors without the need to be physically present. This approach helps increasing operational efficiency and allows centralized operators to manage multiple robots deployed across different locations.

To make teleoperation effective, the human operator should be provided with accurate and real-time information about the robot's environment, allowing safe and efficient robot control.

Head Mounted Displays (HMDs), together with *volumetric videos*, can improve operator's *spatial awareness*, since they provide an immersive experience and allow natural viewpoint changes that are not possible with conventional 2D displays and videos.

The volumetric content is generated on the remote robot and must be streamed to the HMD in real-time, requiring a dedicated *streaming pipeline* suitable for handling the high data rates and low-latency constraints of immersive content delivery.

Volumetric video is typically represented in the form of meshes or point clouds (PCs), both of which are characterized by very high bandwidth requirements [17]. In the context of robot teleoperation, the need for *real-time* data transmission is critical. In fact, latency may hinder the operator's ability to respond promptly, resulting in degraded performance and a less effective control experience. These challenges are further exacerbated by the limited computational resources typically available on ARMs, which constrain the selection of software frameworks to those with low processing requirements, excluding many solutions designed for high-performance workstation and laptop systems.

In this scenario, the rapid deployment of the Fifth Generation (5G) networks represents a key enabler, particularly for the real-time transmission of high-volume data [15]. Compared to previous generations, 5G provides a significantly increased bandwidth [12], enabling higher data rates and reduced latency – both critical for immersive teleoperation applications.

This work addresses the challenge of real-time point cloud transmission for volumetric video streaming in the teleoperation of AMRs. The main contributions of this paper are as follows:

- we design a transmission pipeline for streaming volumetric video, telemetry data, and control commands from remote AMRs to an HMD;
- we introduce a sampling strategy that dynamically adapts the size of the transmitted point cloud to the available network bandwidth, reducing latency while preserving the operator's ability to safely control the robot;
- we build a framework for HMDs that supports rendering of point clouds under real-time constraints.

The proposed system is evaluated through experiments on a real mobile robot, including deployment over a 5G testbed. The sampling strategy is evaluated in different bandwidth conditions, highlighting how it reacts to prevent congestion and latencies. Moreover, we evaluate the *Motion-to-Photon* (MtP) latency of the framework with teleoperation tests across two countries, validating the proposed teleoperation pipeline in real use-cases.

2 Related Work

Teleoperation of remote mobile robots navigating an environment poses several challenges. In order to properly control a robot, the delay between the control input, its actuation, and the visual feedback must remain below 150 ms [13, 1]. This constraint is further exacerbated by the limited computational capabilities of the on-board computing unit. Bandwidth limitations require careful consideration of data transmission volumes, as point cloud data impose impractical bandwidth demands, especially in mobile networks [3, 2, 15]. Consequently, compression and filtering algorithms are required to achieve feasible data volumes.

Although point cloud compression schemes, such as G-PCC [8] and V-PCC [9], provide high compression efficiency and support for both lossy and loss-less compression, they exhibit with high encoding/decoding latency [16]. Google Draco [7] achieves $4\times$ compression rate, but proves to be unsuitable for real-time applications, particularly when executed on embedded devices [11, 3].

In [11], the authors introduce GROOT, an end-to-end volumetric video streaming pipeline addressing efficient decoding for embedded devices. A key difference in our work stems from the main application: teleoperating a robot, which also involves streaming control commands. In this work, we propose a framework that takes advantage of WebRTC's *media track* and *data channel* for the streaming pipeline. The former is designed for real-time audio/video transmission over the Real-time Transport Protocol (RTP), while the latter serves as a bidirectional channel for sending any other type of data over the Stream Control Transmission Protocol (SCTP).

A volumetric video streaming framework using WebRTC to address low latency in videoconferencing is introduced in [6]. However, this approach assumes a static background and a distance-based filter to filter points farther than a given distance, a condition generally encountered in videoconferencing applications but incompatible with teleoperation scenarios. Moreover, their compression relies on Google Draco, which only works well with smaller point clouds [11, 3].

In [16], the authors propose a learning-based compression scheme for adaptive bitrate point cloud streaming. While their system provides efficient compression and adaptive bitrate streaming capabilities, the number of frames in the segments directly impact compression efficiency. Nevertheless, the overall latency prevents the framework from being deployed in teleoperation use-cases.

Notice that it is possible to employ background removal strategies for videoconferencing, significantly reducing the number of points. However, such strategies are inapplicable to the teleoperation of mobile robots. In fact, during navigation, receiving a more detailed 3D environment allows operators to follow safer trajectories and anticipate dynamic obstacles.

3 Teleoperation framework

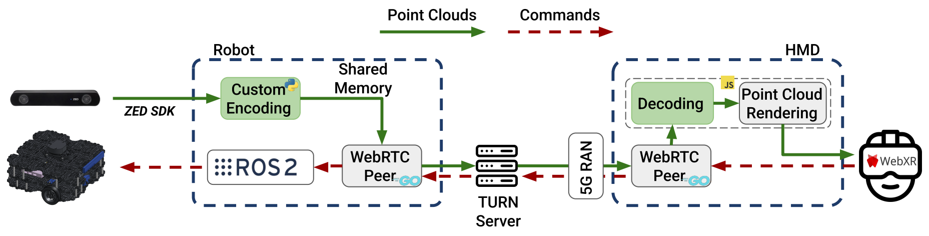


Fig. 1. The proposed immersive teleoperation system

Figure 1 shows the proposed system, which consists of two main components: a mobile robot equipped with an *Nvidia Jetson Xavier NX* and a commercial HMD, the *Meta Quest 3*, connected to the Internet via a 5G network.
The robot's software stack is based on three main components:

- **Robot point cloud producer**: implemented in Python language, this module acquires point clouds from the ZedSDK[2] and applies a filtering and custom encoding strategy to adapt the point cloud bitrate to the measured available bandwidth (see Section 3.2);
- **WebRTC robot peer**: implemented in Golang [10] using the *Pion* WebRTC library [14], this component is responsible for transmitting the encoded point clouds and receiving commands from the remote peer. It interacts with other software onboard using shared memory;
- **ROS2 relay node**: bridges shared memory and ROS2 topics to allow the communication of telemetry and commands to and from ROS2[3] topics;
- **ROS2 navigation nodes**: manage robot locomotion using its navigation stack[4] and handle incoming control commands.

The HMD's software stack consists of two main components:

- **WebRTC HMD peer (Pion Bridge)**: developed in Golang and compiled as a standalone application, it maintains the connection with the WebRTC robot peer, receiving the point cloud stream and transmitting control commands to the robot. It is also referred to as Pion Bridge since it acts as a communication bridge between the peer connection and the UI.

[2] https://www.stereolabs.com/en-it/developers/release
[3] https://github.com/ros2
[4] https://docs.nav2.org/index.html

– **WebXR User Interface**: implemented in Javascript as a single script to maximize performance, this module performs point cloud decoding and rendering in real-time.

In the following, a detailed description of each component is provided. In particular, Section 3.1 describes the WebRTC pipeline, including details about the *Signaling server*. Section 3.2 describes the components onboard the robot that enable the point cloud streaming and the actuation of the received control commands. Finally, Section 3.3 presents the framework running on the HMD which implements the reception, reconstruction, and rendering of point clouds, as well as sending control commands.

3.1 WebRTC pipeline

Signaling. This phase is managed by the *signaling server*, the main component of the WebRTC infrastructure that allows the exchange of connection metadata between the Robot Peer and the Pion Bridge. The signaling server has been implemented in `Node.js` and relies on the WebSocket protocol to support continuous, bidirectional communication between peers. Two WebSockets endpoints are exposed on separate ports: port 3001 for the Robot and port 3002 for the Pion Bridge.

Internally, the server maintains an in-memory data structure that represents its current state. This structure includes the list of registered robots, the Session Description Protocol (SDP) offers sent by the robots, the corresponding answers generated by the clients, and the ICE Candidates exchanged by both parties during the connection negotiation process.
This architecture allows *decoupled and asynchronous* peer management: either the robot or the Pion Bridge may connect first, without compromising the consistency or reliability of the signaling process.

Communication. A WebSocket connection is used to allow communication with the signaling server based on JSON messages. Two main interaction categories are defined: those originating from the robots and those from the client (Pion Bridge).

In particular, the following messages define the interaction between a robot and the signaling server during WebRTC session management:

– *Register*: The robot sends its identifier and the SDP Offer to register with the signaling server.
– *Offer*: The robot updates its SDP offer, typically following a disconnection or renegotiation event.
– *Candidate*: The robot transmits newly discovered ICE candidates to the signaling server for connection establishment.
– *Deregister*: The robot requests its cancellation from the list of active robots to avoid inconsistent states with the Pion Bridge.

The interaction between the Pion Bridge and the signaling server is managed through a defined set of WebSocket messages:

- *getRobots*: The client queries the signaling server for the current list of registered robots and their associated SDP offers.
- *Answer*: The client sends the SDP answer for a specific robot.
- *Candidate*: The client transmits the ICE candidates it has collected for a specific robot.

Connection Lifecycle. The connection process begins with the *robot registration* phase. The robot initiates a WebSocket connection to the signaling server on port 3001 and transmits a `register` message specifying its name and `sdpOffer`. In response, the server assigns a unique identifier (*robotId*) to the robot and returns it.

Next, during the *client connection* phase, the client establishes a WebSocket connection on the port 3002 and retrieves the list of registered robots by sending a `getRobots` message.

Once the list of available robots is obtained, the *SDP negotiation* begins. The client generates an *sdpAnswer* for each robot and sends it to the server, which forwards it to the corresponding robot to complete the WebRTC handshake.

This is followed by the *ICE candidate exchange*. Both the robot and the client communicate their locally discovered ICE candidates to the signaling server using `candidate` messages. These are then relayed to the respective peer to support the establishment of a direct peer-to-peer connection.

In case of a network change or peer disconnection, the robot can update its `sdpOffer` on the signaling server, ensuring consistency and allowing the client to retrieve updated information via a new `getRobots` request to reestablish the connection.

Finally, the *deregistration* phase occurs either explicitly, when the robot sends a `deregister` message, or implicitly, when the WebSocket connection is closed. In both cases, the signaling server removes the robot from the list of active peer.

3.2 The robot's framework

Volumetric content capture Volumetric content is captured onboard the robot in the form of point clouds using the ZED SDK [18], accessed through a custom Python script. The stereo camera is configured to acquire images in HD1080 resolution at a frame rate of 30 FPS. These image frames are processed internally by the ZED SDK to generate point clouds, which are extracted at an arbitrary frequency (in our case 18 FPS). Due to the unified memory architecture of the Nvidia Jetson Xavier NX, each point cloud frame is stored in a $(480 \cdot 480) \times 4$ `Float32` tensor using PyTorch. The first three `Float32` encode 3D spatial coordinates (x, y, z) of each point, while the fourth `Float32` value encodes the RGBA color information. This representation allows for efficient memory usage and compatibility with subsequent processing stages. The resulting point cloud is thus represented by a matrix $\mathbf{P} \in \mathbb{R}^{(480\cdot480)\times4}$, where each row corresponds to a single point, as shown below:

$$\mathbf{P} = \begin{bmatrix}\mathbf{x}\ \mathbf{y}\ \mathbf{z}\ \mathbf{c}\end{bmatrix} = \begin{bmatrix} x_1 & y_1 & z_1 & c_1 \\ x_2 & y_2 & z_2 & c_2 \\ \vdots & \vdots & \vdots & \vdots \\ x_n & y_n & z_n & c_n \end{bmatrix} \tag{1}$$

This volume of data cannot be streamed directly, as it would require a bandwidth of $480 \cdot 480 \cdot 4 \cdot 32 \cdot 18 = 530\,\text{Mbps}$, which exceeds the practical limits of current mobile networks [3, 2, 15]. Therefore, a bandwidth reduction strategy should be designed to enable real-time streaming over mobile connections.

Quantization and filtering To reduce the bandwidth required for point cloud transmission to practical levels, two complementary strategies are adopted: *quantization* and *filtering*.

Quantization: The coordinates of each points' position are quantized to 16 bits. Color information is quantized into a total of 16 bits, distributed across channels as follows: 6 bits for red, 6 bits for green, and 4 bits for blue. With this quantization approach, each point occupies 64 bits, thus reducing the data volume by 50% compared to the original 128-bit representation.

Filtering: Despite quantization, the bandwidth requirements for transmitting full point clouds remain excessive. To further reduce the data rate and adapt it to the time-varying network bandwidth while preserving task-relevant information, a filtering strategy is introduced.

Due to real-time constraints and the limited resources available on typical AMR platforms, traditional compression schemes are often impractical. Therefore, we propose a lightweight distance-based filtering approach, which prioritizes points based on their proximity to the robot.

The following formalizes known parameters. Next, the sampling strategy is introduced.

Known Parameters : The filtering algorithm operates under the following known parameters:

– $n = 480 \cdot 480$ number of points in the raw point cloud $\mathbf{P}$;
– Target frame rate (FPS), specified by the application;
– λ: a tunable parameter that controls the steepness of the distance-based prioritization;
– D: a threshold distance below which all points are prioritized;
– Point size after quantization: 64 bits;
– The *Maximum Frame Size* (MFS), in bits, that can be computed as:

$$MFS = \frac{BW}{FPS} \tag{2}$$

where BW is the estimated available bandwidth.

– The *Maximum Number of Transmittable Points per Frame*, $N_{\max}$:

$$N_{\max} = 150 \cdot \left\lfloor \frac{1}{150} \cdot \max\left(\min\left(\frac{\text{MFS}}{64},\ n \right),\ \frac{0.10 \cdot n}{\text{FPS}} \right) \right\rfloor \tag{3}$$

This expression ensures: 1) that the total number of transmitted points does not exceed the available bandwidth; 2) packet alignment, as the number of points per frame must be a multiple of 150 (matching the packet structure described in Section 3.2) and guaranteeing the transmission of at least 10% of the raw point cloud.

WebRTC, which employs the Google Congestion Control (GCC) algorithm [4], provides feedback on the target bitrate BW in bits per second, reflecting the current network conditions. This target bitrate acts as a system constraint: exceeding it leads to queuing delays and degraded performance [5].

Sampling Strategy: To satisfy these constraints, we apply an importance-based sampling scheme that prioritizes points based on their depth (z-coordinate). This importance value, referred to as *validity*, is defined in Equation 4. The underlying assumption is that points closer to the robot (with smaller z-values) are more relevant for navigation and collision avoidance purposes, while points farther away (with larger z-values) do not represent immediate obstacles.

$$\mathbf{v} = \frac{1}{\left(\frac{\mathbf{z}}{\max(\mathbf{z})}\right)^{\lambda} + \varepsilon}, \quad \mathbf{v} \in \mathbb{R}^{n \times 1} \tag{4}$$

where ε is a small positive constant to avoid division by zero.

The validity values of all points below the guaranteed distance D are then brought to the maximum value of v for all points (Equation 5) to associate them with the highest priority.

$$v_i = \begin{cases} \max(\mathbf{v}) & \text{if } z_i < D \\ v_i & \text{otherwise} \end{cases} \quad \text{for } i = 1, \ldots, n \tag{5}$$

The resulting *validity* is then employed as a priority vector to sample a number $N_{\max}$ of points from the original point cloud using a weighted random sampling implemented in Pythorch using *Pythorch.multinomial*.

Shared Memory To ensure effective, low-latency, and thread-safe communication among these components, a shared memory and semaphore system has been implemented, based on memory-mapped files in `/dev/shm/`, a portion of shared RAM accessible as a filesystem. A total of four memory portions have been instantiated: two for data and two for their corresponding semaphores, to enable synchronization between reading and writing operations.

– `shared_pc`: Transfers the processed point cloud from the Python component to the GoLang component. This is done via a binary buffer with a 3-byte

header, which will contain the data size, and a payload of 1,843,200 bytes ($480 \times 480 \times 8$ bytes, the maximum PointCloud size allowed by the Codec). An header file (`shared_pc_head`) is then used to hold the following information:
  - an access flag, which is set to 1 when a new frame is available in the `shared_pc` memory, while the reading process resets its value to 0 after reading, in order to allow a new writing operation. Notice that this does not act as a semaphore, but rather as a notification flag to enable synchronization between the writing and reading processes.
  - a second flag that indicates the user's intention to perform latency tests on the Codec.
  - the bandwidth obtained from Pion GoLang, as feedback for the PointCloud Codec.

– `shared_control`: Used for sending commands to be executed on the Robot from the GoLang component to ROS2. These commands are represented by a standard `geometry_msgs/Twist` message, which includes two three-dimensional vectors (linear and angular, of type Vector3) and an incremental counter to track the command sequence, for a total of 52 bytes.
  A second synchronization flag, `shared_control_read`, that synchronizes access to the `shared_control` file.

WebRTC robot peer The GoLang component of the Robot's software stack is responsible for establishing the bidirectional connection with the Pion Bridge Peer, through the signaling Server, and for sending and receiving data.

Since point clouds are an unconventional data type and are not natively supported by Pion (the main WebRTC library for Go), a Custom Media Track has been implemented to extend `webrtc.TrackLocal` to transport 3D data via the RTP protocol.

– *Codec and metadata*: To enable Pion to send data without employing any codec, a 90 kHz video/pcm payload type is registered. Each point cloud frame is then divided into RTP packets, to which the following header is attached:
  - `FrameNr` (`uint32`): Sequential frame number;
  - `FrameLen` (`uint32`): Total frame length in Bytes;
  - `SeqOffset` (`uint32`): Offset in Bytes of the current portion;
  - `SeqLen` (`uint32`): Size of the current portion in Bytes;
  - `Data`: Data buffer, up to 1200 Bytes.

  Each packet contains 150 points, each represented by 8 bytes: 6 bytes for the spatial position (`X Y Z` coordinates) and 2 bytes for the color. The final packet size is 1200 Bytes, where the first 900 Bytes correspond to the spatial components (`X Y Z ... X Y Z`) of each point, and the remaining 300 Bytes represent the color data for each point (`C C C ... C`). This *packetization* process is shown in Figure 2.

– *Congestion control and feedback*: Pion's interceptors (send-side GCC, TWCC, NACK, and PLI) are instantiated and added to the Custom Media Track to allow the Python codec to obtain a bandwidth estimate, and to prevent congestion or increased latency;

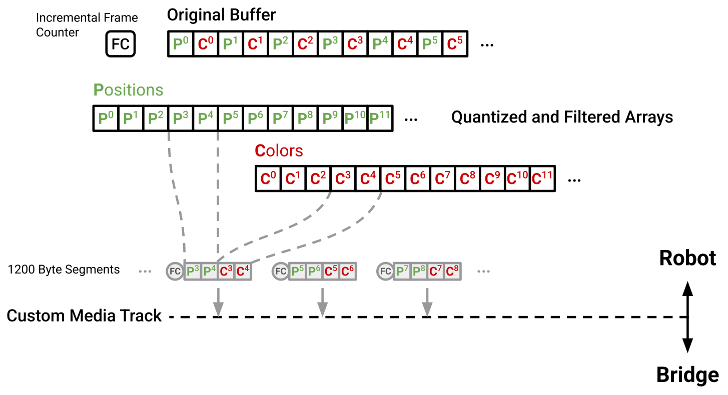


Fig. 2. Point Cloud Frame Packetization

- *Media Track Flow*: On track startup, a dedicated *goroutine* monitors the shared memory. As soon as a new point cloud buffer becomes available, the data is packetized into 1200-byte segments; the header defined above is added to each segment, and then every packet is dispatched. At the end of this process, a current bandwidth estimate is requested from GCC and then relayed to the Python component using Shared Memory.

At startup, along with the Custom Media Track, the following DataChannels are created:

- *Control*: Instantiated by the robot, it is responsible for receiving movement commands to be executed via ROS2. These commands are encoded in JSON (linear and angular fields) and written to shared memory. It is also used to receive the user's intention to perform a latency test;
- *Bandwidth*: Instantiated by the Pion Bridge Peer, it is responsible for receiving 4 Bytes that indicate the bandwidth estimate produced by the other Peer. Although the estimate provided by Pion's GCC implementation is currently used, it may prove useful for future implementations;
- *ControlTrack*: Instantiated by the Pion Bridge Peer, it handles the remote management of RTP transmission (start/stop), via `"start"` and `"stop"` text messages.

3.3 HMD's Software Stack

WebRTC HMD's peer After the *Web-UI* initialization, a connection is established with three distinct WebSocket endpoints by the WebRTC Peer: the *Control Socket*, the *Telemetry Socket* and the *Streaming Socket*. In this way, the WebRTC peer acts as a local *bridge* between the WebRTC streams and the HMD's User Interface.

Concerning the WebSockets, the *Control Socket* (`controlSocket`) manages the control messages between the Web-UI and the HMD's WebRTC Peer. It also

handles the operator's choice of which robot to teleoperate, which in turn defines the source of the point cloud stream.
A key function of the `controlSocket` is the transmission of an initialization command to the local Pion Bridge. This command includes the `signalingURL`, which contains the address of the signaling server. Providing this URL enables the Pion Bridge to establish WebRTC connections. Additionally, this socket requests updates on the list of available robots and updates the UI accordingly. Then, the Pion Bridge routes only the RTP track of the selected robot, temporarily suspending all other connections by sending a *stop command* on the *ControlTrack DataChannel*, effectively reducing bandwidth consumption. Thus, the Pion Bridge practically behaves like a *multiplexer*, switching views and controls between different robots.

The *Telemetry Socket* (`telemetrySocket`) is tasked with sending teleoperation commands from the *Web-UI* to the local Pion Bridge. These commands, generated from user inputs on the HMD, are then relayed by the Pion Bridge to the selected remote robot.

The *Streaming Socket* (`streamingSocket`) receives the point cloud packets. To prevent the degradation of one connection from affecting all others, the Pion Bridge allows the establishment of multiple concurrent WebRTC connections in distinct processes. After receiving the point cloud packets, the Pion Bridge reconstructs the coordinates and colors buffers (see Section 3.2) and sends it to the point cloud rendering pipeline (Section 3.3). *Point cloud frame reconstruction*: To optimize the aggregation process, two separate buffers are pre-allocated: one for coordinates, whose size is 900 Bytes, and one for colors, occupying 300 Bytes. As each packet arrives, its coordinates and color data are appended to the corresponding buffers, as shown in Figure 3. This process is presented in Figure 3.

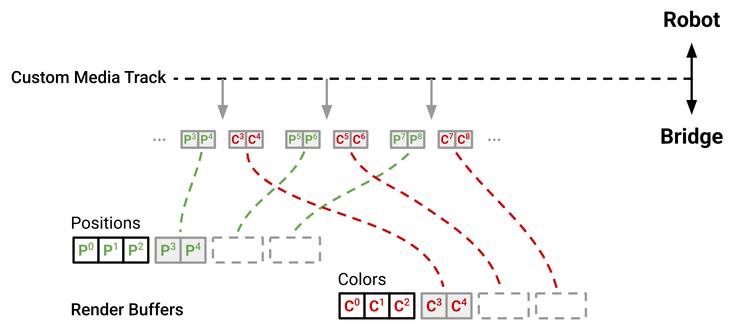


Fig. 3. Point Cloud Frame Depacketization

After the full point cloud frame is received, these buffers are combined and resized proportionally to the number of points received in the frame.
The final structure is a buffer consisting of the coordinates for the first 75%

and the colors for the remaining 25%. The buffer is then sent to the *point cloud rendering pipeline* to be rendered on the *Web-UI.*

Point cloud rendering pipeline The HMD's WebRTC peer calculates the number of points using the buffer size. Then, using the byte offset, the data type and the data size, it translates and casts the point cloud frame into a *typed array.* This approach minimizes the access to memory, as typed arrays always refer to the same memory address, and avoids unnecessary computation.

Then, leveraging *ThreeJS*, a library that enables the use of typed arrays, three-dimensional geometries are defined through the use of the *Buffer Geometry* class[5]. This kind of 3D objects have their own attributes that are then used by the rendering pipeline, implemented using WebGL[6], to draw the geometry onto the canvas. In this step, the position and color attributes are updated to match the ones received. This forces a geometry update and allows for the rendering of incoming frames. The main advantage of using WebGL is represented by the wide range of data types supported by the shader programs involved in the rendering step. This allows for a seamless implementation of encoding strategies based on quantization.

To efficiently render the point cloud, a custom `ShaderMaterial` is employed to avoid any unnecessary calculation and ensuring the best visual quality. The custom ShaderMaterial is done through a vertex and a fragment shader in OpenGL Shading Language (GLSL), responsible of reading the attributes from the buffers and using them to render each individual point with the right position, color and dimension on the screen.
The vertex shader is employed to calculate the final screen position of each point and implementing an *adaptive sizing mechanism.* In particular, it dynamically adjusts the rendered size of each point, `gl_PointSize`, to enhance depth perception and visual clarity specifically for dense point clouds. The adaptive sizing logic is designed to offer nuanced visual control. In particular, let p_{base} be the size of points; s_{obj} the scale of the point cloud in the scene; and z_{local} the z coordinate of the point with respect to the origin of the point cloud; then, each point's size `gl_PointSize` is calculated as follows:

$$\text{gl\_PointSize} = p_{base} \cdot \frac{s_{obj}}{w_{clip}} \cdot (1 + |z_{local} \cdot f_{scaleZ}|)$$

where w_{clip} estimates the distance between the user's point of view and each point; f_{scaleZ} is a sensitivity parameter to control how significantly the point's Z-position affects its final rendered size;
Note that the term $\frac{s_{obj}}{w_{clip}}$ applies perspective scaling, making points further from the camera appear smaller, and accounts for the scale of the object in the WebGL scene, ensuring the final screen size of the point scales properly when manipulating the point cloud object. The term $(1 + |z_{local} \cdot f_{scaleZ}|)$ refines the size of each point, ensuring that p_{base} is the minimum point size.

[5] https://threejs.org/docs/#api/en/core/BufferGeometry
[6] https://developer.mozilla.org/en-US/docs/Web/API/WebGL_API

The proposed scaling technique provides valuable depth cues related to the object's internal structure, supplementing the standard perspective projection. For optimal performance, this computation is executed within the vertex shader, leveraging the parallel processing capabilities of the GPU. Moreover, points are rendered as circles instead of squares to enhance the visual quality.

At runtime, the parameters for adaptive point sizing are determined by empirical functions. These functions were initially proposed and later qualitatively assessed to strike an optimal balance between simplicity and the visual quality of the experience.

Notice that the point cloud elaborations are executed inside a `WebWorker`, while the rendering process is handled by the main thread of the browser. The two components communicate through messages handled by callbacks and support zero-copy buffer transmission, vital to satisfy the latency requirements. This separation ensures a fluid user experience, reducing motion sickness that could otherwise be caused by dropped frames.

Teleoperation Commands XR controllers are used to send commands to the remote AMR. The teleoperation commands are represented in linear and angular velocity values derived from the controller's thumbstick: the linear command is set based on the y-axis value, while the angular command is derived from the x-axis value.

Once a teleoperation command is generated, a throttling check is performed to ensure that commands are sent at a fixed rate. Then, commands are sent to the Pion Bridge via the `telemetrySocket`.

4 Results and Discussions

In this section we evaluate the key components and provide insights on the proposed framework. In particular, Section 4.1 presents the tests conducted on the sampling strategy under different network conditions and λ values; in Section 4.2, in order to provide insights on the performance of the framework as a whole, we evaluate the Motion-to-Photon (MtP) latency of the framework, defined as the sum of the latency for sending the user command and the latency to receive the new point cloud from the robot; Section 4.3 provides key insights on the framework's performance with respect to industrial contexts and VR comfort requirements.

4.1 Sampling results

In this section, we evaluate the performance of the sampling strategy in different bandwidth conditions and λ values. For the purpose, we fix $D = 3.85$ m and FPS $= 10$ Hz.

Sampling with different λ values Figure 4 presents the effects of different bandwidth conditions on the sampling strategy when fixing different λ values. It is shown how lower λ values lead towards a uniform sampling strategy, which hinders the prioritization during sampling of all points closer than D (see Figure 4(a)). Higher λ values, instead, benefit the prioritization of those points, effectively matching the original distribution for distances below D at the expenses of further points (see Figure 4(c)).

Sampling in different bandwidth conditions Then, we test different λ values under three different bandwidth conditions.

When the bandwidth prevents the streaming of points closer than D, then the distribution of retained points will be scaled down to prevent congestion (i.e. critical delays). In these conditions, when λ assumes higher values, the retained points after D are dramatically reduced.
It is important to note that, since we are using a weighted probability distribution to sample the point cloud, the distribution of sampled points still depends on the distribution of the original point cloud. In particular, the lower λ, the higher this dependency is accentuated. This effect can be seen in Figure 5, where setting $\lambda = 0$ results in having a uniform *validity* for all distances. When applied to the raw point cloud distribution, it follows the same trend, but scaled down to satisfy the constraint on the $N_{\max}$ points to be sent. On the contrary, as Figure 5(c) shows, higher λ values retain the (scaled down, due to lower bandwidth) original distribution for all distances lower than D, effectively prioritizing those points with respect to those at a distance higher than the threshold.

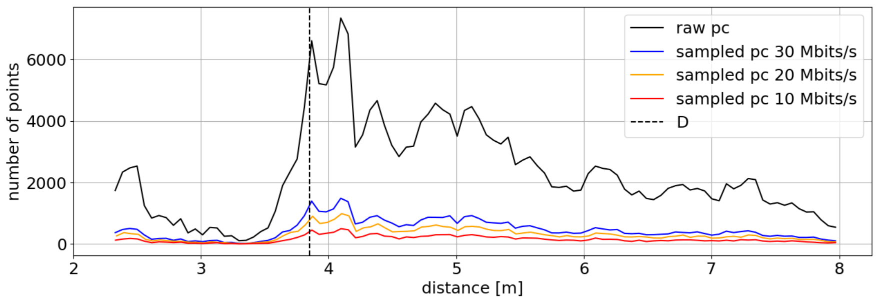
4.2 Results on the testbed

Latency test In order to measure the *Motion-to-Photon* (MtP) delay between the moment the user applies a control input and the moment the new point cloud frame is received, we embed a flag in the command message and store its timestamp. Once the robot receives the control command with the flag, it will turn the next point cloud frame color green. Then, on the HMD's side, once a frame of green points is received, its timestamp is saved and compared with the control command's one. Moreover, for testing purposes, the green point cloud will still be rendered. This provides the user with a visual hint of the actual MtP delay.

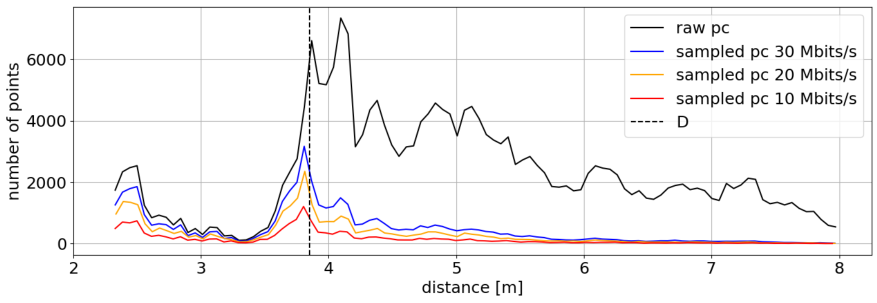
In order to simulate real use cases, for this test the human operator (with an HMD) and the robot, are deployed 1748.5 kilometers apart in straight-line distance. In particular, the user is located at the University of Surrey (UK) and the Robot at Politecnico di Bari (Italy). The average available link bandwidth, measured with *iperf3*[7], is 41.44 Mbit/s, while the average RTT is 42.4 ms.

Figure 6 shows two experiments with different bandwidths conditions. The figure shows the measured latency as well as the number of points transmitted.

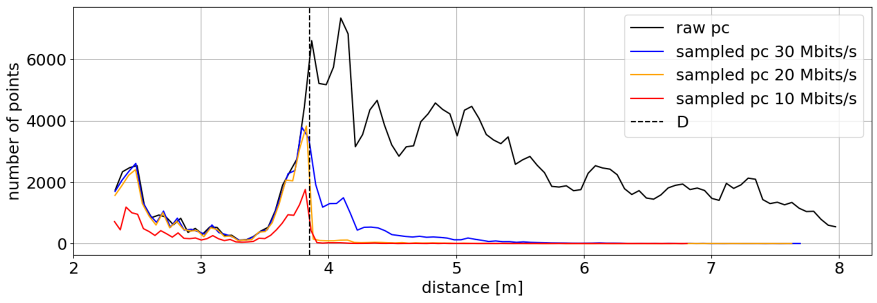
[7] https://iperf.fr/iperf-download.php



(a) Test 1 - $\lambda = 0$

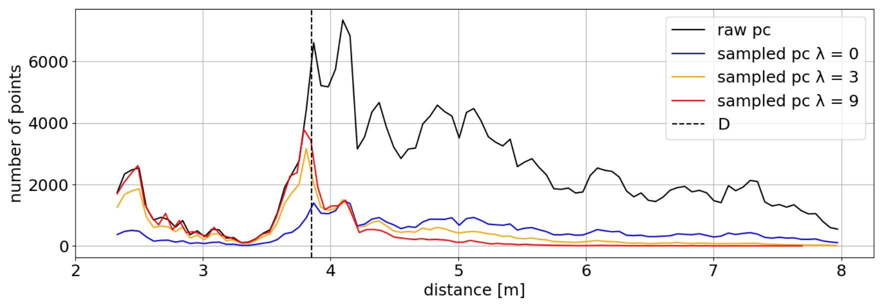


(b) Test 2 - $\lambda = 3$

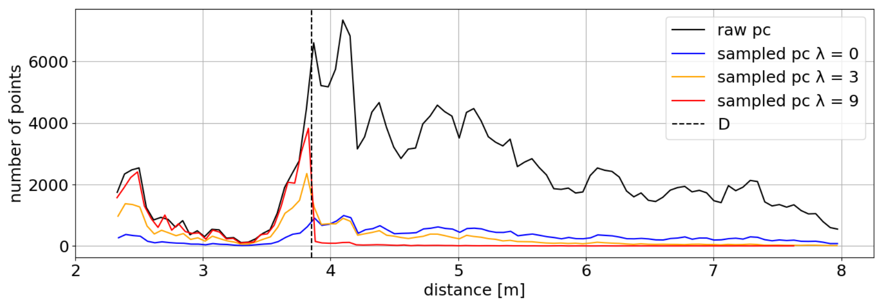


(c) Test 3 - $\lambda = 9$

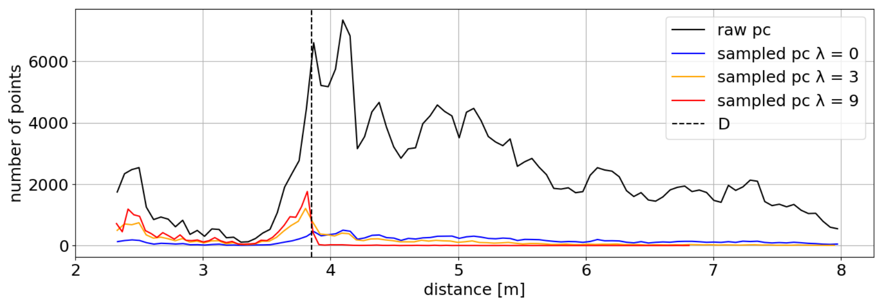
Fig. 4. Sampling Strategy with different λ values



(a) Test 1 - $BW = 30$

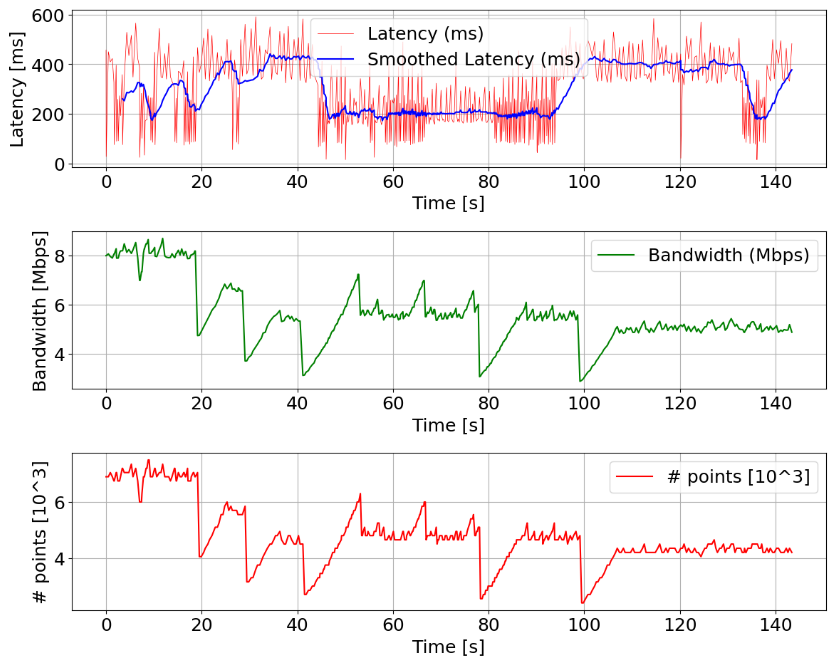


(b) Test 2 - $BW = 20$

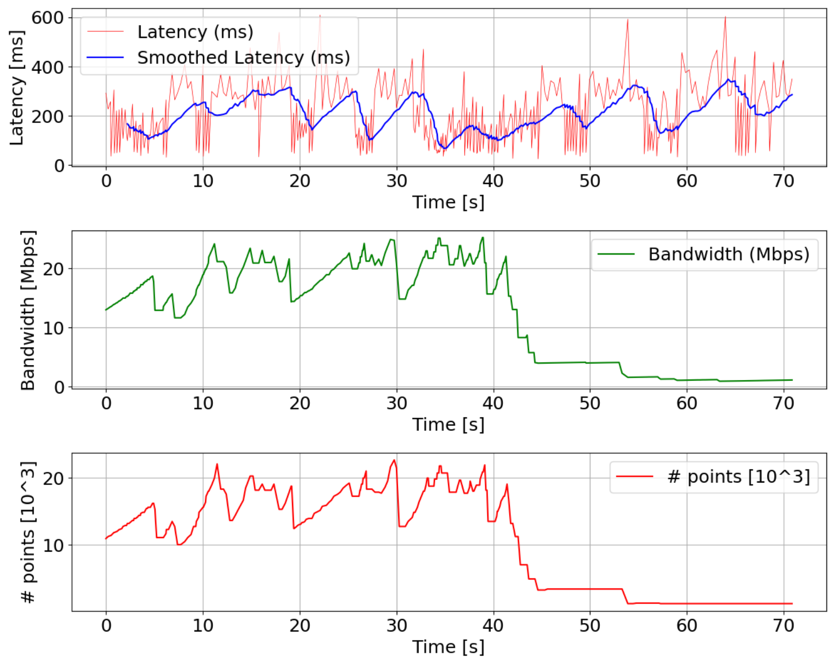


(c) Test 3 - $BW = 10$

Fig. 5. Sampling Strategy in different bandwidth conditions



(a) Test 1



(b) Test 2

Fig. 6. Network tests

The latter is used to provide information on both the reaction of the adaptation system to bandwidth changes and the quality of the received point cloud (where a higher number of points is associated with better quality).

Throughout the tests, the latency remains under 400 ms, with spikes associated to sudden drops in bandwidth. Note that such result is to be expected, since the framework modulates the number of points to be sent proportionally to the measured bandwidth. This effectively prevents high latencies at the expenses of the point cloud density, enabling the operator to timely stop the robot if the bandwidth hinders the streaming of a satisfying number of points.

4.3 Discussions

A key feature of the rendering pipeline is the decoupled point cloud elaboration - rendering process. Using different threads for handling and rendering point clouds received enables the viewport to always be reactive and responsive to the user inputs (i.e. head movements and viewport manipulation). This separates the point cloud capture and reception FPS from the viewport FPS, alleviating well-known VR issues of motion sickness and improving overall operator comfort. In fact, the viewport FPS is only related to the TreeJS performance on the HMD. Moreover, in the worst case where point clouds are not received at all, the user head movement will still produce a movement in the viewport. In order to thoroughly assess such aspects, we plan to extend this work with user experience evaluations through, e.g., motion sickness assessments (Simulator Sickness Questionnaire) and operator workloads (NASA Task Load Index). These evaluations will serve as starting point for further improvements on the proposed framework.

5 Conclusions

In this paper, we proposed a framework for immersive teleoperation of AMRs, introducing a real-time volumetric video streaming pipeline. Leveraging both quantization and an adaptive sampling strategy, the framework is able to adapt to different bandwidth conditions, ensuring low latencies even in low bandwidth conditions.

Tests conducted show that the proposed distance-based sampling strategy effectively prioritizes relevant information for navigation purposes. Furthermore, employing the WebRTC for transmission allowed for real-time interaction with the robot, keeping averages latencies compatible with the practical teleoperation requirements, especially in 5G enabled networks.

Further tests could be conducted to study how the proposed framework affects the Quality of Experience (QoE) during operations. Moreover, the sampling strategy could be further optimized to better react to bandwidth oscillations.

6 Acknowledgments

This work has been funded by the European Union through the DIANE project winner of the Open Call n.1 of the SPIRIT project (101070672).

References

1. 5GAA: Tele-operated driving (tod): System requirements, analysis and architecture (2020), https://5gaa.org/tele-operated-driving-tod-system-requirements-analysis-and-architecture/, accessed: 2025-01-18
2. Barone, N., Brescia, W., Mascolo, S., De Cicco, L.: Apeiron: a multimodal drone dataset bridging perception and network data in outdoor environments. In: Proceedings of the 15th ACM Multimedia Systems Conference. p. 401–407. MMSys '24, Association for Computing Machinery, New York, NY, USA (2024). https://doi.org/10.1145/3625468.3652186, https://doi.org/10.1145/3625468.3652186
3. Barone, N., Brescia, W., Santangelo, G., Maggio, A.P., Cisternino, I., De Cicco, L., Mascolo, S.: Real-time point cloud transmission for immersive teleoperation of autonomous mobile robots. In: Proceedings of the 16th ACM Multimedia Systems Conference. p. 311–316. MMSys '25, Association for Computing Machinery, New York, NY, USA (2025). https://doi.org/10.1145/3712676.3719263, https://doi.org/10.1145/3712676.3719263
4. Carlucci, G., De Cicco, L., Holmer, S., Mascolo, S.: Congestion control for web real-time communication. IEEE/ACM Transactions on Networking **25**(5), 2629–2642 (2017). https://doi.org/10.1109/TNET.2017.2703615
5. De Cicco, L., Carlucci, G., Mascolo, S.: Congestion control for webrtc: Standardization status and open issues. IEEE Communications Standards Magazine **1**(2), 22 – 27 (2017). https://doi.org/10.1109/MCOMSTD.2017.1700014
6. De Fré, M., van der Hooft, J., Wauters, T., De Turck, F.: Scalable mdc-based volumetric video delivery for real-time one-to-many webrtc conferencing. In: Proceedings of the 15th ACM Multimedia Systems Conference. pp. 121–131. Association for Computing Machinery, New York, NY, USA (2024)
7. Google: Google draco, https://github.com/google/draco
8. Group, M.P.E.: Information technology - coded representation of immersive media - part 9: Geometry-based point cloud compression (g-pcc) (2023), https://www.iso.org/standard/78990.html, published March 2023
9. Group, M.P.E.: Information technology - coded representation of immersive media - part 5: Visual volumetric video-based coding (v3c) and video-based point cloud compression (v-pcc) (2025), https://www.iso.org/standard/89030.html, published March 2025
10. Inc., G.: The go programming language, https://go.dev/
11. Lee, K., Yi, J., Lee, Y., Choi, S., Kim, Y.M.: Groot: a real-time streaming system of high-fidelity volumetric videos. In: Proceedings of the 26th Annual International Conference on Mobile Computing and Networking. MobiCom '20, Association for Computing Machinery, New York, NY, USA (2020). https://doi.org/10.1145/3372224.3419214, https://doi.org/10.1145/3372224.3419214
12. Lv, J., Lin, Y., Hou, M., Li, Y., Gao, Y., Dong, W.: Accurate bandwidth and delay prediction for 5g cellular networks. ACM Trans. Internet Technol. **25**(2) (Apr 2025). https://doi.org/10.1145/3703629, https://doi.org/10.1145/3703629

13. Musicant, O., Botzer, A., Shoval, S.: Effects of simulated time delay on teleoperators' performance in inter-urban conditions. Transportation Research Part F: Traffic Psychology and Behaviour **92**, 220–237 (2023). https://doi.org/https://doi.org/10.1016/j.trf.2022.11.007, https://www.sciencedirect.com/science/article/pii/S1369847822002753
14. Pion: Pion webrtc, https://github.com/pion/webrtc
15. Raca, D., Leahy, D., Sreenan, C.J., Quinlan, J.J.: Beyond throughput, the next generation: a 5g dataset with channel and context metrics. In: Proceedings of the 11th ACM Multimedia Systems Conference. p. 303–308. MMSys '20, Association for Computing Machinery, New York, NY, USA (2020). https://doi.org/10.1145/3339825.3394938, https://doi.org/10.1145/3339825.3394938
16. Rudolph, M., Rizk, A.: Learned compression in adaptive point cloud streaming: Opportunities, challenges and limitations. In: Proceedings of the 16th ACM Multimedia Systems Conference. p. 328–334. MMSys '25, Association for Computing Machinery, New York, NY, USA (2025). https://doi.org/10.1145/3712676.3719266, https://doi.org/10.1145/3712676.3719266
17. Schwarz, S., Preda, M., Baroncini, V., Budagavi, M., Cesar, P., Chou, P.A., Cohen, R.A., Krivokuća, M., Lasserre, S., Li, Z., et al.: Emerging mpeg standards for point cloud compression. IEEE Journal on Emerging and Selected Topics in Circuits and Systems **9**(1), 133–148 (2018)
18. Stereolabs: Stereolab zed sdk - using the depth sensing api, https://www.stereolabs.com/docs/depth-sensing/using-depth#getting-point-cloud-data