Proceedings of the International Conference on
Formal Methods and Foundations of Artificial Intelligence
Eszterházy Károly Catholic University
Eger, Hungary, June 5–7, 2025
pp. 188–200
DOI: 10.17048/fmfai.2025.188

Benchmarking data logging strategies in ROS-integrated multi-sensor robots under network constraints

Nour Elhouda Ben Saadi[a], **Zoltán Istenes**[b]

Eötvös Loránd University
nourelhoudabensaadi@gmail.com
istenes@inf.elte.hu

Abstract. This work benchmarks multiple logging strategies, including JavaScript Object Notation (JSON) serialization, JSON Lines (JSONL), WebSocket based rosbridge, native Robot Operating System (ROS) Transmission Control Protocol (TCP)/User Datagram Protocol (UDP) transports, and NFS-based recording, on a multisensor robot over a 100 Mbps constrained network. We evaluate dropped messages, resource usage, and storage footprint across high-throughput data streams, such as LiDAR, Red-Green-Blue plus Depth (RGB-D) images, Inertial Measurement Unit (IMU), and Global Navigation Satellite System (GNSS) data. Results show that TCPROS provides the highest reliability, while UDPROS, rosbridge, and JSON-based methods incur significant losses when network capacity is saturated. Network File System (NFS) recording performs well, provided that network stability is maintained. Our findings reveal key trade-offs between transport guarantees, network limitations, and message size fragmentation.

Keywords: ROS, data logging, benchmarking, TCPROS, UDPROS, JSON, multisensor robots, network constraints

AMS Subject Classification: 68T40, 68M10, 68M20

1. Introduction

Accurate and timely data logging in robotic systems is crucial for tasks such as performance evaluation, fault diagnosis, and developing machine learning pipelines. As robots increasingly integrate high-throughput sensors – such as Light Detection and Ranging (LiDAR), RGB-D cameras, inertial units, and GNSS modules – the abil-

ity to reliably record data is critical for both real-time decision-making and offline analysis. Recent advancements in ROS-based architectures, including modular sensor integration and networked communication protocols, have improved scalability and flexibility in robotic software systems. However, in bandwidth-constrained or infrastructure-limited environments – such as field robotics or industrial inspection – data transport can become a significant bottleneck. Under such conditions, large message sizes and unreliable links often cause dropped messages, increased Central Processing Unit (CPU) overhead, and inconsistent logging performance. Efforts like ROS Enhancement Proposal (REP)2014 propose benchmarking guidelines for ROS 2 systems to standardize performance analysis under diverse computational and network conditions [5]. The RobotPerf suite [4] further enables reproducible benchmarking of computational workloads in ROS 2 pipelines. Additionally, recent studies on ROS 2 real-time latency and transport performance [2, 12] demonstrate the value of systematic evaluations across middleware configurations. However, despite ROS 2 gaining popularity, ROS 1 remains widely deployed across academic and industrial settings, especially in long-lived platforms and legacy deployments. While most recent benchmarks focus on ROS 2, there is a lack of in-depth evaluation of logging strategies in ROS 1, particularly under constrained network conditions, where transport choice, message encoding, and system load directly affect data reliability. To address this gap, we conduct a comprehensive benchmarking study of ROS 1 data logging strategies in multi-sensor robotic systems, focusing on real-world limitations caused by limited bandwidth.

Our key contributions are as follows:

- We benchmark six data logging strategies in ROS 1: TCPROS, UDPROS, rosbridge (WebSocket), JSON serialization, JSONL, and NFS-based remote logging.
- We evaluate each strategy in a real-world multi-sensor setup (LiDAR, RGB-D, IMU, GNSS) across a 100 Mbps constrained network.
- We compare performance based on message loss, CPU and memory usage, and storage efficiency.
- We analyze trade-offs related to transport guarantees, serialization overhead, and system saturation under network stress.

In this paper, we provide a comprehensive evaluation of data logging strategies in ROS 1 under constrained network conditions on a multi-sensor robotic platform. Unlike prior studies, our benchmarks are performed in real time during active sensor streaming, capturing the actual performance limitations encountered in deployment scenarios. The remainder of this paper is structured as follows: Section 2 discusses related work. Section 3 presents the system architecture and logging strategies. Section 4 details the benchmarking methodology. Section 5 presents the results and their analysis. Finally, Section 6 concludes and outlines future directions.

2. Related work

Benchmarking in robotic systems has gained increased attention with the evolution of modular and distributed architectures, particularly in ROS 2. The official benchmarking guideline, REP-2014, provides recommendations for reproducible performance analysis in ROS 2 systems, including tracing, latency measurement, and profiling [5]. Building on this, the RobotPerf suite offers a vendor-neutral benchmarking framework for computational performance in ROS 2, targeting CPU usage, scheduling behavior, and system latency under sensor load [4]. However, both focus on ROS 2 middleware and compute pipelines rather than message transport or data logging behavior. Several other works have examined ROS 2 communication performance. Kronauer et al. analyzed latency behavior in multi-node Data Distribution Service (DDS) based systems and highlighted the trade-offs between configuration settings and timing guarantees [2]. Yanlei et al. conducted a comparative evaluation of ROS 1 and ROS 2 real-time performance under CPU load, noting ROS 2's architectural improvements in timing determinism [12]. Still, these studies do not explore logging reliability or network-induced message loss. In the ROS 1 ecosystem, tools like ROS-CBT focus on communication benchmarking by measuring throughput and latency over emulated network links between virtual nodes [3]. While ROS-CBT includes tests under constrained bandwidth, it does not evaluate end-to-end logging mechanisms or analyze serialization overhead, system load, or message loss during real-time recording. Furthermore, it is simulation-based and does not reflect the challenges of logging live, high-throughput sensor data on deployed robotic systems. Other works like ROSfs propose user-space file systems for sharing ROS data across distributed robots, but they do not benchmark performance or logging efficiency under network constraints [11]. To the best of our knowledge, no prior work has systematically evaluated data logging strategies in ROS 1 under constrained network conditions using a live, multi-sensor robotic platform. Our study fills this gap by benchmarking six strategies – including TCPROS, UDPROS, and NFS recording, as well as rosbridge-based logging of ROS messages in raw ROS Bag file format for logging messages (rosbag) format, JSON, and JSONL. We evaluate message loss, CPU and memory usage, and storage footprint under real-time conditions with active sensor streaming.

3. System description

Our evaluation was conducted on a ROS 1-based robotic platform – we named Pomona – designed for sensor-rich navigation and mapping tasks. The robot we used is an Agile-X four-wheel-drive Scout Mini[1] robot equipped with the Research Pro sensor kit (see Figure 1).

The system integrates a mix of high- and low-throughput sensors, covering a broad range of data rates including:

[1] https://global.agilex.ai/products/scout-mini

Figure 1. Pomona: a ROS 1-based multi-sensor robotic platform used in our experiments. The system integrates a Velodyne VLP-16 LiDAR, an Intel RealSense D435 RGB-D camera, an Xsens IMU, and a GNSS receiver.

- **LiDAR**: Velodyne VLP-16 (`/velodyne_points`, `sensor_msgs/PointCloud2`), 10 Hz, average bandwidth ~6.59 MB/s, average message size ~0.62 MB
- **RGB-D camera**: Intel RealSense D435
  - **Color image** (`/camera/color/image_raw`, `sensor_msgs/Image`): 30 Hz, average size ~0.92 MB, bandwidth ~27–28 MB/s
  - **Depth image** (`/camera/depth/image_raw`, `sensor_msgs/Image`): 30 Hz, average size ~0.81 MB, bandwidth ~25 MB/s
  - **Compressed color** (`/camera/color/image_compressed`, `sensor_msgs/CompressedImage`): 30 Hz, average size ~16.5 KB, bandwidth ~510 KB/s
  - **Compressed depth** (`/camera/depth/image_compressed`, `sensor_msgs/CompressedImage`): 30 Hz, average size ~22 KB, bandwidth ~670 KB/s
- **IMU**: Xsens MTI 600 series (`/imu/data`, `sensor_msgs/Imu`), 25 Hz, message size ~0.32 KB, bandwidth ~8 KB/s
- **GNSS receiver**[2]: (`/gnss`, `sensor_msgs/NavSatFix`), 4 Hz, message size ~124 bytes, bandwidth ~640 B/s
- **Base controller**: Scout Mini (`/scout_status`, `scout_msgs/ScoutStatus` [1]), 50 Hz, message size ~0.14 KB, bandwidth ~7.3 KB/s

[2]Incorporated into the Xsens IMU

These sensors interface with an onboard NVIDIA Jetson AGX Xavier running ROS Melodic (Ubuntu 18.04), streaming data over a 100 Mbps Ethernet link to an external logging workstation (Ubuntu 22.04). This constrained bandwidth emulates real-world field robotics and industrial inspection scenarios, where infrastructure or power limitations restrict data transmission. Under these conditions, we evaluated the following six logging strategies.

3.1. TCPROS (TCP)

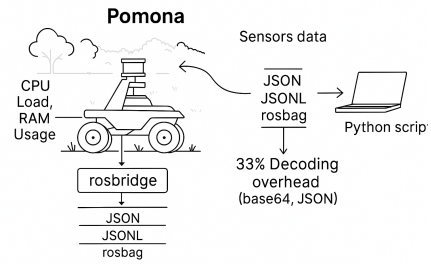
TCPROS [6] is the default ROS 1 transport, using TCP for reliable, in-order delivery between publishers and subscribers. As illustrated in Figure 2b, Pomona's sensor nodes publish topics over the 100 Mbps Ethernet link to a *Docker* container running on the logging workstation. The container uses a ROS 1 image (e.g., `ros:melodic-ros-base`) to provide the correct environment for `rosbag record`. This containerized setup is necessary because the logging workstation runs Ubuntu 22.04, which is incompatible with ROS Melodic, and it avoids installation conflicts or dependency issues on the host system [9, 10]. Host networking (`-network host`) and a bind mount to `/data` enable direct topic subscription and efficient data storage. Each publisher–subscriber pair establishes a persistent TCP channel. TCP handles retransmissions and enforces ordering, ensuring message integrity for high-bandwidth topics such as LiDAR scans, RGB-D images, and IMU readings. Under high network utilization, TCP back-pressure increases ROS publish queues, and if these queues overflow, messages are dropped at the publisher before transmission.

3.2. UDPROS (UDP)

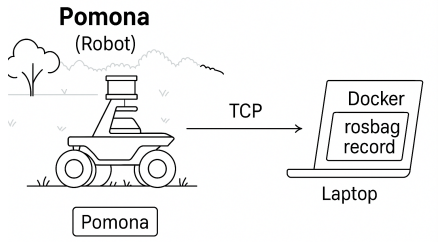
UDPROS [7] is a connectionless ROS 1 transport that uses the User Datagram Protocol (UDP) to transmit messages without delivery guarantees. As illustrated in Figure 2c, Pomona's sensor nodes publish topics over the 100 Mbps Ethernet link to a *Docker* container on the logging workstation running a ROS 1 Melodic image. Direct use of `rosbag record` could not successfully negotiate the UDP transport layer with Pomona's publishers, as ROS 1 defaults to TCPROS unless both endpoints explicitly advertise UDPROS support. To enable UDP-based recording, we implemented an intermediate C++ relay node inside the container. This node subscribed to Pomona's topics via UDPROS, confirmed using `rostopic info`, and re-published the messages locally within the container. `rosbag record` then subscribed to these local topics, benefiting from intra-process communication while preserving the UDP characteristics of the original inbound link.

3.3. rosbridge + rosbag

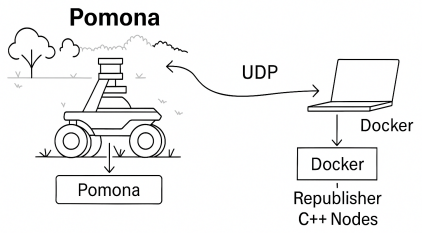
The `rosbridge_server` [8] exposes ROS topics over a JSON/WebSocket interface. As illustrated in Figure 2a, Pomona runs `rosbridge_server` and streams messages to the logging workstation over WebSocket. On the workstation, we



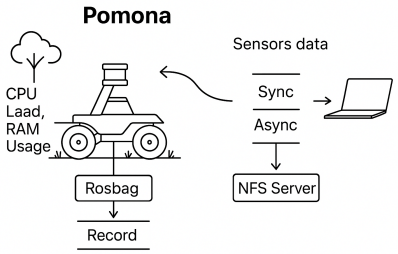
(a) ROSBridge Architecture



(b) TCP Architecture



(c) UDP Architecture



(d) NFS Architecture

Figure 2. Comparison of architectures: ROSBridge, TCP, UDP, and NFS.

run a *Docker* container with a ROS 1 Melodic image to provide the required environment for `rospy`, `rosbag`, `genpy`, and message definitions. Inside the container, a Python bridge (via `roslibpy`) subscribes to the WebSocket topics, reconstructs native ROS messages, and writes them to a `.bag`. For message types with binary payloads (`sensor_msgs/CompressedImage`, `sensor_msgs/PointCloud2`), `rosbridge_server` transmits the `data` field as base64 text; we therefore *decode to raw bytes prior to message construction*. Without this step, the bag appears valid in `rosbag info` but the payload bytes are incorrect, leading to corrupted images and point clouds at playback.

3.4. rosbridge + JSON

In this configuration (Figure 2a), Pomona's onboard ROS 1 Melodic system runs `rosbridge_server` to serialize ROS messages into JSON and stream them over a WebSocket connection. The logging workstation connects via `roslibpy` from a Python script and saves incoming messages directly to a JSON file for each topic. The subscriber script maintains an in-memory dictionary keyed by topic name and appends each received message before periodically writing to disk. For topics containing binary payloads (e.g., `sensor_msgs/Image`, `sensor_msgs/CompressedImage`, `sensor_msgs/PointCloud2`), the raw JSON representation can produce very

large files and, if not explicitly base64-encoded before transmission, may risk truncation or corruption depending on client library handling. This approach bypasses `rosbag record` entirely, allowing collection on systems without a full ROS environment, but requires careful handling of binary fields to ensure data integrity.

3.5. rosbridge + JSONL

In this configuration (Figure 2a), Pomona runs `rosbridge_server` to serialize ROS messages into JSON objects, which are transmitted over a WebSocket connection to the logging workstation. Instead of aggregating messages into a single file, each incoming message is written as an individual JSON object on its own line (JSON Lines, or JSONL format). The subscriber script, implemented in Python with `roslibpy`, appends each received message to a text file as soon as it arrives. This streaming write pattern enables incremental storage without keeping the entire dataset in memory and simplifies parsing for large-scale post-processing, since each line is a self-contained JSON object. As with the pure JSON approach, topics containing binary fields (e.g., `Image`, `CompressedImage`, `PointCloud2`) require base64-encoding of their `data` fields to avoid payload corruption. Without decoding these fields back into raw bytes during reconstruction, the resulting files will appear structurally valid but contain unusable image or point cloud data.

3.6. NFS recording

In this configuration (Figure 2d), the logging workstation mounts a remote directory from Pomona via the Network File System (NFS) protocol. On Pomona, `rosbag record` runs natively within the ROS 1 Melodic environment, writing directly to the mounted path, which physically resides on the workstation's storage. This arrangement keeps all ROS topic subscription, serialization, and bag creation on the robot side, while the workstation receives the bag file writes in real time through the NFS link. It avoids the need for a ROS environment on the workstation for recording and guarantees full compatibility with Pomona's Melodic setup. However, overall performance is bound by NFS throughput and the robot's disk I/O; high-rate or large-payload topics may still cause write delays or dropped messages if the network or storage becomes saturated.

4. Benchmarking methodology

All strategies were tested in real time while the robot continuously published sensor data. Logging scripts were implemented in Python and C++ where applicable. Depending on the strategy, performance metrics were collected in the robot or in the external logging machine. These metrics include CPU load, memory usage, storage footprint, and message drop rate. While Linux tools such as `htop`, `iftop` were consulted for runtime inspection, we primarily used `time -v` to obtain detailed per-process performance metrics at critical evaluation points during logging.

To evaluate the performance of each logging strategy under bandwidth- and resource-constrained conditions, we conducted real-time experiments with the robot Pomona while it actively streamed high-rate sensor data. Each strategy was launched independently while keeping the sensor configuration constant to ensure comparability. For each strategy, we performed at least three independent runs, each lasting 120 seconds, to capture representative performance and account for run-to-run variability. Performance data were recorded concurrently on either the onboard or the external logger, depending on where the logging process was executed. Metrics measured included:

- **CPU usage**: peak and average utilization of the logging process, using: `time -v`
- **Memory consumption**: maximum Resident Set Size (RSS), from `time -v`.
- **Bandwidth utilization**: observed using `iftop` on the robot's Ethernet interface (note: not measured separately for each logging strategy).
- **Storage footprint**: file size of logged data.
- **Message drop rate**: estimated based on comparing the expected message counts, computed as:

$$\text{Expected Count} = f_{nominal} \times T_{recording} \tag{4.1}$$

  where $f_{nominal}$ is the nominal topic frequency and $T_{recording}$ is the logging duration, with the actual received counts in the logs. The drop percentage was then calculated as:

$$\text{Drop Rate (\%)} = \frac{\text{Expected Count} - \text{Actual Count}}{\text{Expected Count}} \times 100 \tag{4.2}$$

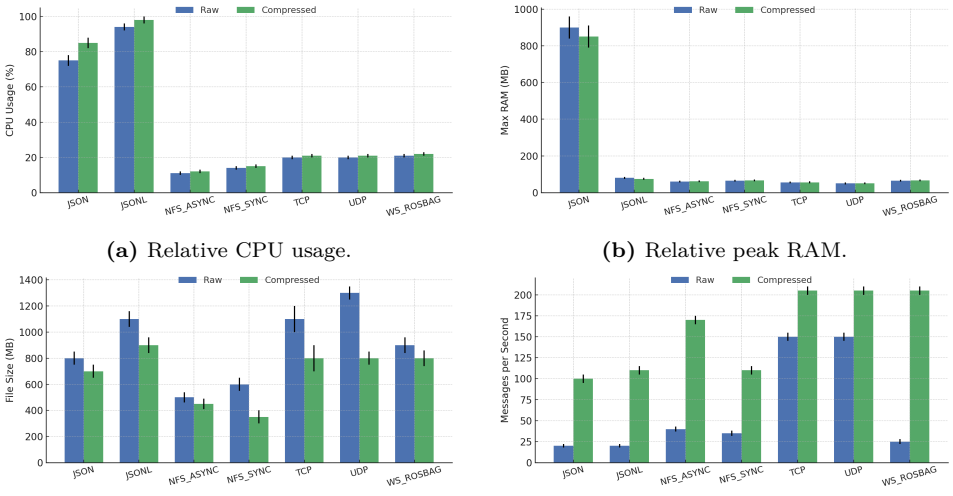
  This method allowed us to estimate the proportion of received messages relative to the expected volume, providing a quantitative basis for drop rate analysis.

The `time -v` outputs were saved automatically for each run and processed using a custom script to generate tabular data, which were then read into Pandas data frames for plotting and further analysis. For fairness, background system load was kept minimal during all runs. For both persistent-mode experiments and standard runs that completed successfully without anomalies, all reported metrics are presented as the mean together with the standard deviation, computed across multiple experimental repetitions.

5. Results and analysis

5.1. System performance and throughput

JSON vs JSONL. Figure 3 compares key performance metrics for all logging strategies under raw and compressed configurations. CPU usage (Figure 3a) shows



(a) Relative CPU usage.

(b) Relative peak RAM.

(c) Relative output file size.

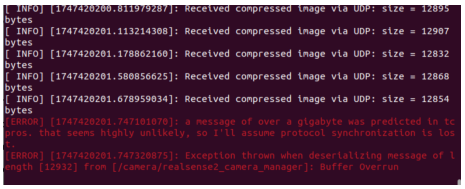
(d) Relative message throughput.

Figure 3. Relative system load and throughput metrics for raw and compressed configurations across all methods.

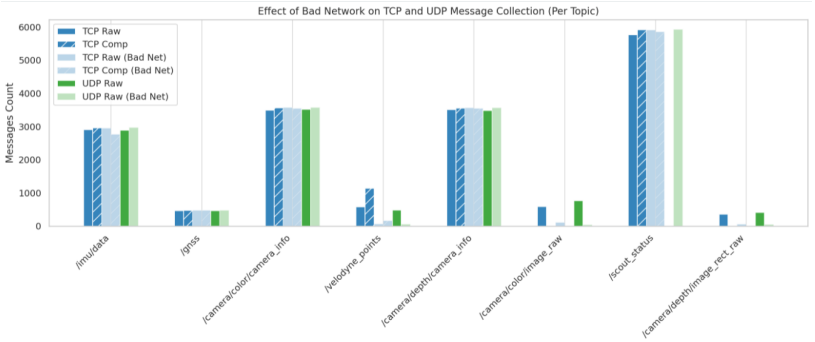
that JSONL, while still exhibiting high CPU load due to its intensive disk I/O, maintains the low RAM usage (Figure 3b). This allows continuous recording without memory saturation, in contrast to the JSON strategy where unsaved data accumulates in RAM until a flush to disk is triggered. Under long runs, this progressive growth (shown in Figure 6a) continues until the output file size exceeds host-level limits, at which point logging halts with an `Errno 27: File too large` error and incomplete data persistence (Figure 4a). To further investigate JSONL's behavior, we experimented with tuning its buffering size (Figure 6c). While smaller buffers kept message throughput relatively stable over extended runs, larger buffers initially increased the sustained rate but eventually led to a sudden drop after prolonged operation. This effect is attributed to accumulated write delays and internal queue growth, which cause bursts of backpressure when the disk subsystem is momentarily saturated. The behavior, although less severe than JSON's abrupt halts, still indicates a limit to how long JSONL can maintain peak throughput without periodic flushing or additional I/O optimization.

Native ROS transports (TCPROS, UDPROS) and NFS-based logging maintain low CPU and RAM usage, with minimal differences between raw and compressed operation. File size results (Figure 3c) highlight the trade-off between storage footprint and message throughput (Figure 3d). Compression reduces output size across all methods, but throughput generally drops for more serialization-heavy approaches. A notable exception is the `ws_rosbridge_rosbag` method, which performs poorly on raw data but shows throughput in the compressed configuration comparable to TCPROS and UDPROS. This counter-intuitive result – achieving

(a) JSON method halting due to file-size limitation (`Errno 27`).



(b) UDP logging failure during compressed image streaming (`Buffer Overrun`).



(c) Network instability affecting both TCP and UDP transports, causing simultaneous camera and LiDAR dropouts until Wi-Fi reset.

Figure 4. Observed runtime failures and instability. Top: method-specific logging errors (a: JSON, b: UDP). Bottom: link-level instability affecting native ROS transports (c).

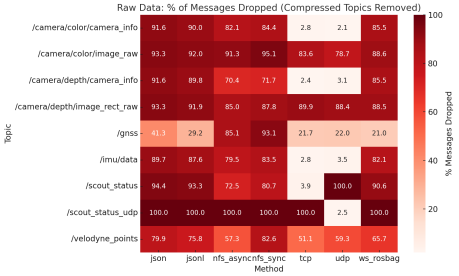
similar throughput while producing smaller files – suggests that compression mitigates bottlenecks in the WebSocket and `rosbag` pipeline, allowing data to be processed and written at a higher sustained rate.

UDP instability and MTU sensitivity. Beyond the JSON limitations, UDP exhibited run-time instability that forced a hybrid setup. When large payloads (camera frames/LiDAR packets) approached or slightly exceeded the link MTU, the UDPROS stream intermittently stalled. Lost fragments caused transport desynchronization, manifesting as spurious size predictions and `Buffer Overrun` errors in the camera node (Figure 4b). Increasing socket and ROS-level buffers reduced drops for small and medium topics, but did not prevent stalls for large payloads under link instability. During affected runs we restored operation by republishing locally (UDP relay → local topics). As a practical mitigation, lowering image resolution/quality or enabling camera-side compression reduces packetization pressure; otherwise TCPROS is preferable for large messages on unstable links. Given this fragility, the Reliability results in Section 5.2 correspond to a *hybrid transport configuration*: UDPROS was retained for low-bandwidth, latency-sensitive topics (IMU, GNSS, status), while large-payload topics (camera frames, LiDAR scans)

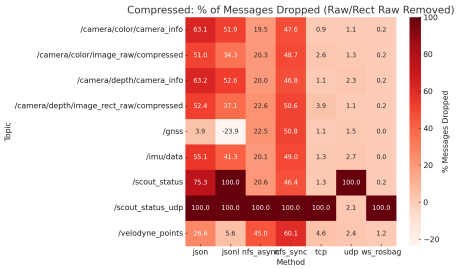
were switched to TCPROS to prevent fragmentation stalls and ensure reliable delivery under variable link conditions. As shown in Figure 6b, this hybrid approach preserved the high delivery rate for small topics while restoring near-complete reception for large payloads.

General network instability. In addition to protocol-specific issues, several experimental runs in both TCP and UDP experiments, intermittent Wi-Fi link throttling impacted performance, where throughput dropped sharply even though the link did not fully disconnect. This manifested as sudden throughput drops and stalled topics certainly over heavy connections, and in some cases required manually resetting the wireless interface to recover (Figure 4c). While rare, such events highlight the importance of robust reconnection and buffering logic in real-world field deployments.

5.2. Reliability: Message Delivery

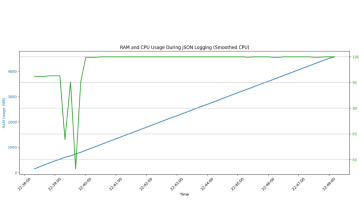


(a) Raw data. Compressed topics omitted.

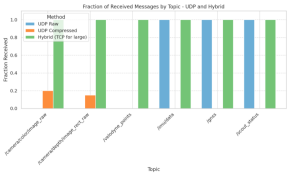


(b) Compressed data. Raw image topics omitted.

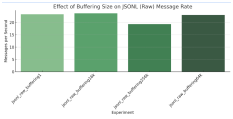
Figure 5. Percentage of messages *dropped* per topic and method for raw vs. compressed data streams.



(a) RAM and CPU usage during JSON logging.



(b) Fraction of received messages for UDP raw vs hybrid transport.



(c) Effect of buffering size on JSONL (raw) message rate.

Figure 6. Additional runtime observations: (a) CPU and RAM growth in JSON logging; (b) improvement in message delivery using hybrid UDP/TCP; (c) impact of buffer size on JSONL throughput.

Drop rates. Figures 5a–5b report message loss as Equation 4.2 indicates. On

raw streams (Figure 5a), JSON/JSONL exhibit consistently high loss across topics, while native transports (TCPROS, UDPROS) retain most control/metadata messages (e.g., `camera_info`, `imu/data`) but still struggle with raw image topics. The dedicated UDP status topic (`/scout_status_udp`) is correctly preserved under UDPROS, matching the transport design. With compression enabled (Figure 5b), TCPROS, UDPROS, and `ws_rosbag` achieve near-zero loss across the board, whereas JSON/JSONL still drop substantially on high-rate sensors. We also observe occasional $> 100\%$ "received" artifacts (visible as negative drop in the source percentage plots) on JSONL channel corresponding to the `/gnss` topic, suggesting message duplication on the WebSocket path rather than true reliability gains. Overall, compression shifts the balance strongly in favor of native ROS transports and the `ws_rosbag` pipeline for dependable delivery under the constrained link.

6. Conclusion and future work

This study provided a systematic benchmarking of ROS 1 data logging strategies under varying payload sizes, compression settings, and network conditions. By evaluating CPU usage, memory footprint, storage requirements, and message delivery rates, we mapped the trade-offs that influence transport and storage performance in real-world deployments. Throughput-oriented transports such as TCPROS, UDPROS, and `ws_rosbag` achieved the highest message rates, particularly with compression enabled. However, UDP's strong performance on small, latency-sensitive topics was offset by its fragility under network stress, making hybrid strategies – where large payloads fall back to TCP – more robust. NFS-based logging, especially asynchronous NFS, offered excellent CPU efficiency but struggled with sustained high-frequency data streams. WebSocket-based JSON/JSONL methods – while easy to set up and flexible – incurred heavy CPU load and high drop rates on large data streams due to serialization overhead. Across all methods, raw (uncompressed) configurations were more vulnerable to packet loss when network capacity was saturated.

No single method proved universally optimal across all conditions tested: the best choice depends on application constraints such as CPU availability, network stability, required throughput, and tolerance for packet loss. Our results provide empirically grounded guidance for selecting or combining transports to meet specific operational requirements in ROS 1-based robotic systems.

Future work will integrate precise time synchronization (e.g., NTP/chrony or hardware timestamping) to enable accurate end-to-end latency analysis. We also plan to extend this benchmarking framework into a more automated, deployment-agnostic tool that can isolate bottlenecks, profile shared resources, and adapt transport selection in real time based on topic characteristics and network state. Additional research will address challenging domains such as real-time LiDAR compression, power and thermal budgeting for edge deployments, validation in mobile, unstable wireless environments, and a hybrid migration strategy in which the work-

station transitions to DDS-based ROS 2 for improved resilience, while the robot remains on ROS 1 and communicates via rosbridge for interoperability. These steps will move toward adaptive logging systems that dynamically balance throughput, resource usage, and reliability in diverse field robotics applications.

References

[1] AgileX Robotics: *Scout_msgs ROS Package*, Accessed: 2025-08-10, 2025, URL: https://github.com/agilexrobotics/scout_ros/tree/master/scout_msgs/msg.

[2] T. Kronauer, J. Pohlmann, M. Matthe, T. Smejkal, G. Fettweis: *Latency Analysis of ROS2 Multi-Node Systems*, arXiv preprint arXiv:2101.02074 (2021), https://arxiv.org/abs/2101.02074.

[3] K. Masaba, A. Quattrini Li: *ROS-CBT: Communication Benchmarking Tool for the Robot Operating System: Extended Abstract*, in: Aug. 2019, pp. 1–3, DOI: 10.1109/MRS.2019.8901094.

[4] V. Mayoral-Vilches, J. J. Jabbour, Y.-S. Hsiao, et al.: *RobotPerf: An Open-Source, Vendor-Agnostic Benchmarking Suite for Evaluating Robotics Computing System Performance*, arXiv preprint arXiv:2309.09212 (2023), https://arxiv.org/abs/2309.09212.

[5] V. Mayoral-Vilches, I. Lütkebohle, C. Bédard, R. Araújo: *REP-2014: Benchmarking performance in ROS 2*, ROS Enhancement Proposal, https://ros.org/reps/rep-2014.html, 2022.

[6] Open Robotics: *ROS TCPROS Transport*, Accessed: 2025-08-10, 2025, URL: http://wiki.ros.org/ROS/TCPROS.

[7] Open Robotics: *ROS UDPROS Transport*, Accessed: 2025-08-10, 2025, URL: http://wiki.ros.org/ROS/UDPROS.

[8] Open Robotics: *rosbridge Suite*, Accessed: 2025-08-10, 2025, URL: http://wiki.ros.org/rosbridge_suite.

[9] O. Robotics: *ROS Distributions and Ubuntu Compatibility*, Accessed: 9 August 2025, 2025, URL: https://wiki.ros.org/Distributions.

[10] O. Robotics: *ROS Melodic Morenia*, Accessed: 9 August 2025, 2023, URL: https://wiki.ros.org/melodic.

[11] S. Schneegass, P. Weiss: *ROSfs: A File System for Robot Operating System Based Data Exchange*, arXiv preprint arXiv:2406.10635 (2024), https://arxiv.org/abs/2406.10635.

[12] Y. Yanlei, Z. Nie, X. Liu, F. Xie, Z. Li, P. Li: *ROS2 Real-time Performance Optimization and Evaluation*, Chinese Journal of Mechanical Engineering 36.1 (2023), p. 144, DOI: 10.1186/s10033-023-00976-5.