Bridging ROS 2 and GNU Radio for Connected Robotics

German Felipe Sanchez Solano GSANCHEZSOLA2019@FAU.EDU
George Sklivanitis GSKLIVANITIS@FAU.EDU
Dimitris A. Pados DPADOS@FAU.EDU
Center for Connected Autonomy and AI (CA-AI.fau.edu), Florida Atlantic University, Boca Raton, FL, USA

Abstract

Connected robotic teams require reliable links under diverse radio conditions, yet data distribution service (DDS) adopted in the Robot Operating System (ROS 2) as the underlying communication layer limits robots to only IP-compatible commercial off-the-shelf radio links. This work presents an IP-free link adapter that transports native ROS 2 topics over GNU Radio by packetizing serialized messages that are in common data representation (CDR) format into fixed-size protocol data units (PDUs) with compact headers, topic identifiers, and fragmentation support. We build custom out-of-tree GNU Radio blocks compatible with ROS 2 Humble and GNU Radio 3.10 to support wireless broadcast delivery of ROS topics data using existing PHY-layer designs in GNU Radio such as orthogonal-frequency-division multiplexing (OFDM). We experimentally evaluate the new GNU Radio blocks on ROS 2 odometry data that are exchanged between two simulated robotic agents through a direct loopback simulation in GNU Radio. We measure end-to-end latency, packet delivery ratio, goodput, and demonstrate in a custom-built GUI how communication errors affect the fidelity of the received trajectory data By decoupling the robotics middleware from the data transport, our new GNU Radio blocks enable systematic experimentation with modulation/coding, framing, and scheduling across heterogeneous PHY/MAC/NET prootocol stacks for connected robotics applications.

1. Introduction

Connected robotic systems perform many complex tasks through coordination, such as cooperative search of an environment, consensus, rendezvous, and formation control. At their core, these systems rely on local coordination between intelligent agents making reliable, low-latency, high-rate wireless communication of primary importance. Beyond simply maintaining connectivity, reliable communication may mean supporting heterogeneous and possibly time-varying communication rates among different pairs of agents. For example, some agents may need to use the network for transmitting video while others may simply wish to transmit status information. The default Data Distribution Service (DDS) communication stack (Morita & Matsubara, 2018) in ROS 2 exhibits significant performance degradation over lossy wireless links. DDS is a brokerless, peer-to-peer architecture based on the Real-Time Publish-Subscribe (RTPS) protocol. DDS provides fine-grained Quality of Service (QoS) configurations to support various robotic workloads, including sensor data streaming, control commands, and state feedback (Kang et al., 2012). The Zenoh middleware integration is a runtime middleware (RMW) plugin for ROS 2 that provides an alternative to the traditional DDS-based communication layer. Instead of relying on the DDS standard that underlies most default RMW implementations in ROS 2 (e.g., Fast DDS, Cyclone DDS), `rmw_zenoh` (Project, 2025) builds on the Zenoh protocol, which is designed for data-centric, distributed communication spanning pub/sub, queries, and storage.

Commodity radio chipsets offer little control over the physical layer (PHY), medium-access-control (MAC) and network (NET) (Macenski et al., 2022; omg, 2019) of the protocol stack. Software-defined radios (SDRs) offer flexibility and open-access to parameters across layers of the wireless protocol stack. Despite the widespread use of ROS 2, integration of ROS 2 traffic to custom protocol stacks for ad-hoc connected robotics experimentation remains unexplored. DDS can operate over custom protocol stacks as long as they are IP-based and support the required QoS semantics (reliability, deadlines, durability). Zenoh can operate over both IP and non-IP links.

In this paper, we describe an approach to bridge ROS 2 and GNU Radio to enable the integration of both existing and future out-of-tree GNU Radio projects for software-defined wireless mesh networking, with the ROS ecosystem for connected robotics applications. Our work is mo-

Proceedings of the 15^{th} GNU Radio Conference, Copyright 2025 by the author(s).

tivated by the NSF-sponsored research testbed at FAU's Center for Connected Autonomy and AI, which focuses on mmWave networked robotics. To demonstrate robot-to-robot communication, we incorporate OFDM-based software transceivers that are available in GNU Radio. ROS-GNU Radio integration is achieved through two custom GNU Radio blocks that interface the PHY-layer OFDM blocks with CDR-formatted data from ROS2. We retain ROS 2's native CDR serialization to preserve message semantics. More specifically, we packetize the CDR payloads into fixed-size PDUs, convey them over to a PHY-layer transmitter implemented in GNU Radio and reconstruct the original ROS messages at the receiver. Decoupling ROS 2 topics from DDS' IP constraints enables systematic experimentation with custom PHY/MAC/NET protocol stacks while preserving native ROS message types. Our goal is to separate robotics middleware from transport to enable the usage of non-IP links for experimental evaluation and development with SDRs. We experimentally evaluate the new GNU Radio blocks on ROS 2 odometry data that are exchanged between two simulated robotic agents through a direct loopback simulation in GNU Radio. We measure end-to-end latency, packet delivery ratio, goodput, and demonstrate in a custom-built GUI how communication errors affect the fidelity of the received trajectory data By decoupling the robotics middleware from the data transport, our new GNU Radio blocks enable systematic experimentation with modulation/coding, framing, and scheduling across heterogeneous PHY/MAC/NET prootocol stacks for connected robotics applications.

2. Related Work

ROS and ROS 2. ROS established the de facto standard for robotics software by using a modular publish/subscribe model (Quigley et al., 2009). ROS 2 is a more modern, robust, and distributed architecture which relies on DDS/RTPS for discovery and transport (Macenski et al., 2022; omg, 2019) for real-time communication, improved multi-robot and embedded platform support. In most cases, DDS uses UDP messages for the underlying communication over the network. While DDS provides robust, industry-standard networking capabilities, it is designed for controlled IP network environments. This can cause problems in experimental wireless ad-hoc networks, which are dynamic and unreliable. Our work aims to fill that gap by keeping ROS serialization but bypassing IP/DDS.

Bridging ROS over web/IP. *rosbridge* provides a JSON/WebSocket gateway to ROS for non-ROS clients and browsers (Crick et al., 2017). By encapsulating ROS messages in IP-based protocols (such as WebSockets, HTTP/REST, or WebRTC), robots can interact not only within local networks but also with cloud services, browser-based dashboards, and remote operators. This approach enhances scalability and accessibility by allowing developers to use standard Internet infrastructure. It is IP-based and optimized for web interoperability rather than wireless ad-hoc network experimentation.

IP-compatible Protocol Stacks on SDRs. Related work in the literature that implements 802.11-compliant protocol stacks in SDRs include *Sora* which demonstrated in real-time 802.11 on commodity PCs (Tan et al., 2009) and *OpenWiFi* that offers a full-stack 802.11a/g/n implementation on Zynq SoCs compatible with Linux `mac80211` (Jiao et al., 2020). Projects such as srsRAN (formerly srsLTE) provide a full LTE/LTE-A protocol stack, including PHY through radio resource control layers, integrated with the Linux networking stack for IP connectivity (Gomez-Miguelez et al., 2016). Similarly, OpenAirInterface (OAI) delivers an end-to-end open-source LTE/5G NR stack for USRP-based SDRs, bridging SDR-based radios with IP-enabled core networks (Nikaein et al., 2014).

Multi-robot Communications. The tight coupling between communications and task performance and the need for experimentation beyond commodity radios in multi-robot systems is emphasized in (Gielis et al., 2022). An example of SDR-enabled collaborative autonomy is the ERA reference application (Sisbot et al., 2019), which demonstrated multi-vehicle map fusion using ROS and the GNURadio-based implementation of the 802.11p (Bloessl et al., 2013).

Contributions. In this paper, we introduce an IP-free protocol that provides a compact wire format for multiplexing and fragmenting ROS 2 messages into fixed-size PDUs in GNU Radio. To support this functionality, we developed two GNU Radio OOT blocks: the ROS Sink `ros_rx` , which translates ROS messages into PDUs, and the ROS Source `ros_tx` , which converts PDUs to ROS messages. These blocks also support a topic registry, namespace mapping, and configurable MTU settings. To guide traffic prioritization, the design incorporates a lightweight QoS policy, implemented as a 2-bit field in the header that enables latency-versus-throughput trade-offs. The system is evaluated through loopback tests with channel models and software-defined radio experiments using OFDM, with preliminary results confirming feasibility while highlighting the overhead and loss trade-offs involved.

3. Background on ROS 2

ROS 2 is an open-source middleware and toolchain for building *distributed* robot software. Applications are composed of *nodes*—independent processes that communicate by publishing and subscribing to *topics*. A topic

Field	Type	Meaning (units)
`header.stamp`	time	Source timestamp (secs, nsecs) set by the publisher.
`header.frame_id`	string	Coordinate special frame (e.g., `map`, `odom`).
`pose.position`	float[3]	(x, y, z) position in meters.
`pose.orientation`	float[4]	Quaternion (x, y, z, w) (unitless) for 3D orientation.
`twist.linear`	float[3]	Linear velocity (v_x, v_y, v_z) in m/s.
`twist.angular`	float[3]	Angular velocity $(\omega_x, \omega_y, \omega_z)$ in rad/s.

Table 1: ROS 2 `nav_msgs/Odometry`: The structure of a topic carrying odometry data.

is a named, unidirectional bus for streaming messages. Publishers and subscribers are decoupled in space (no addresses exchanged), time (late joiners can subscribe), and synchronization (asynchronous delivery). Namespaces (e.g., `/robot_1`) are used to provide a scope for sets of topics per robot and prevent name collisions, particularly within multi-robot deployments.

In ROS 2 Humble, discovery, data dissemination, and QoS management are provided by DDS. DDS organizes communication and enforces QoS contracts (e.g., reliability, durability, deadline, liveliness, history), while participant discovery and on-wire interoperability are handled by the RTPS protocol. Message payloads are serialized using CDR, and DDS/RTPS typically transports them over UDP/IP. Each communication topic is strictly associated with a specific, pre-defined data type. This ensures that any data sent by a publisher on a given topic is structurally identical to the data that subscribers expect to receive.

A topic is a typed stream of messages. Messages have fixed schemas (e.g., odometry, LiDAR). Many include a timestamp embedded and other relevant metadata to ensure seamless communication between nodes at the application layer. Examples of messages of topics `/odom` and `/scan` with their structures are shown in Fig. 1 and Fig. 2. These messages are sent at pre-configured rates by their respective nodes.

Odometry */odom* ROS type `nav_msgs/Odometry` represents a robot's pose and velocity at a moment in time. A topic such as `/robot_1/odom` is a *stream* of these messages at rate R Hz. As summarized in Table 1, an *Odometry* message typically reads values in the structure shown in Fig. 1. LaserScan */scan* ROS type `sensor_msgs/LaserScan`) represents a 2D LiDAR scan as an array of distances measured at fixed angular increments. Figure 2 shows a parsed message, as described in Table 2.

```
header:
  stamp:
    {sec: 174518, nanosec: 649000000}
  frame_id: "map"
pose:
  position:
    {x: 1.23, y: -0.45, z: 0.00}
  orientation:
    {x: 0.0, y: 0.0, z: 0.38, w: 0.92}
twist:
  linear:
    {x: 0.40, y: 0.00, z: 0.00}
  angular:
    {x: 0.00, y: 0.00, z: 0.10}
```

Figure 1: Example of ROS 2 `nav_msgs/Odometry` message truncated and decoded in human-readable text.

```
header:
  stamp:
    {sec: 174519, nanosec: 120000000}
  frame_id: "laser"
angle_min: 0.0
angle_max: 6.283185307
angle_increment: 0.01745
range_min: 0.12
range_max: 12.0
ranges:
  [ 1.98, 2.03, 2.01, ... ]
intensities:
  [ 1600.0, 1620.0, 1612.0, ... ]
```

Figure 2: Example ROS 2 `sensor_msgs/LaserScan` message truncated and decoded in human-readable text.

4. ROS 2-GNU Radio System Overview

Figure 3 outlines the proposed *IP-free* data path between ROS 2 serialization and a GNU Radio PHY-layer flowgraph. Our proposed GNU Radio OOT blocks expose message ports carrying PDUs (PMT `dict` + `u8vector`) and preserves ROS types end-to-end. The proposed GNU Ra-

Field	Type	Meaning (units)
`header.stamp`	time	Source timestamp of the scan.
`header.frame_id`	string	Sensor frame (e.g., `laser`).
`angle_min/max`	float	Start/end angles (radians).
`angle_increment`	float	Angular step between beams (radians).
`range_min/max`	float	Valid measurement bounds (meters).
`ranges`	float[N]	Distance for each beam (meters), length N.
`intensities`	float[N]	(Optional) return strength per beam.

Table 2: ROS 2 `sensor_msgs/LaserScan`: The structure of a topic carrying LaseScan (LiDAR) data.

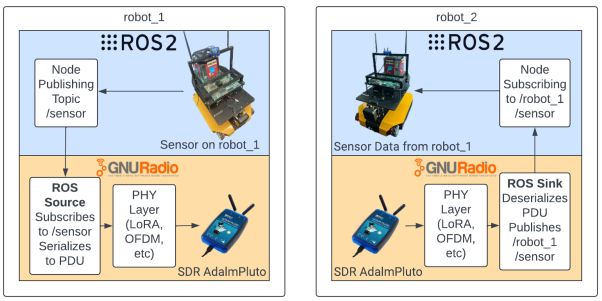

Figure 3: Proposed architecture: ROS→PDU source, GNU Radio PHY, PDU→ROS sink. Message ports carry PDUs (PMT `dict+u8vector`); packetization uses a fixed header, packed Topic IDs, entry records, PAD, and CRC.

dio block are used for both loopback and over-the-air experiments with COTS SDRs that are supported by the GNU Radio ecosystem. The following section describe in more detail the implementation of ROS Source and ROS Sink blocks in GNU Radio.

4.1. ROS Source (ROS→PDU)

Registry & Topic IDs. At initialization, the source block reads a CSV registry (see Table 4) of {`topic_base`, `type`} pairs and assigns `Topic IDs` in `0x0,...,0xF` (nibbles). The registry limits the multiplexed set and provides a stable `tid` for each topic.

Subscriptions. A ROS 2 node is created and subscribers are instantiated for the configured topics (optionally filtered by a runtime `topics` list). Subscriber depth is user-configurable.

Serialization. Each incoming ROS message is serialized using native ROS 2 CDR. The block does not alter message content or headers; `header.stamp` remains intact and is used for latency measurements.

Notation	Definition
$P_{\max}$	Fixed packet size (bytes), including PAD and CRC.
E	EntryCount per packet, $E \in (1,...,15)$.
k_i	Payload bytes of the i-th entry (excludes 5 B header).
B	Byte budget for all entries $B = P_{\max} - \big(7 + \lceil E/2 \rceil\big)$.
S	Sum of entry sizes $S = \sum_{i=1}^{E}(5 + k_i)$.
L_{pad}	PAD length $L_{\text{pad}} = P_{\max} - \big(5 + \lceil E/2 \rceil + S + 2\big) \geq 0$.
Seq	Per-source sequence number (mod 256).
Hop	Hop count (TTL), TTL $\in (1,...,15)$.
Q	QoS policy (2 bits) in `Version/Flags`.
tid	Topic ID (nibble) from the registry (`0x0,...,0xF`).
frag_id	Entry stream identifier (byte) for one message.
idx, *total*	Entry index and total EntryCount.

Table 3: Symbols used in the wire format and analysis.

Topic	Type
`scan`	`sensor_msgs/LaserScan`
`odom`	`nav_msgs/Odometry`
`imu`	`sensor_msgs/Imu`
`cmd_vel`	`geometry_msgs/Twist`
`map`	`nav_msgs/OccupancyGrid`
`clock`	`rosgraph_msgs/Clock`

Table 4: Example topic registry.

Packetization. Serialized bytes are split into *entries* subject to the configured MTU $P_{\max}$ and a minimum entry payload $k_{\min}$. For each PDU, a fixed 5 B header is transmitted: `Version/Flags` (with a 2-bit QoS policy), `Src ID`, `Dst ID` (`0xFF` denotes broadcast), `Seq`, and `Hop|EntryCount`. The list of `Topic IDs` is packed as nibbles (as shown in Table 5). Each entry carries a 5B entry header (Entry ID, Entry Index, Total Entries, Entry Length). `PAD` bytes fill to $P_{\max}$, followed by `CRC`.

PDU Metadata. The PMT `dict` includes minimally the `src_id`, `dst_id`, `seq`, and optional diagnostics. If present, a wall-clock `t_src_wall` supports robust latency calculations.

Transmission to GNU Radio. The PDU is published on a GNU Radio *message port* to the downstream framer/PHY-layer Tx chain.

Transmission Over-the-Air. Any GNU Radio PHY-layer flowgraph (e.g., OFDM, GFSK, LoRa) can transport the PDUs. The proposed ROS Source and Sink blocks are agnostic to framing, modulation, coding, and rate control; such policies are left to the PHY/MAC blocks. The QoS policy in `Version/Flags` is optional, i.e., it may be used to prioritize at the packetizer or ignored by the PHY.

4.2. ROS Sink (PDU→ROS)

Received PDUs are CRC-verified and headers parsed (flags, addressing, sequence, entry count, Topic IDs). Fragments are buffered per (`src_id`, `tid`, `frag_id`) until `idx` reaches `total`. A per-message timeout is used to collect incomplete messages in lossy channel conditions. Upon completion, the payload is deserialized using the registered ROS type for `tid`. The original `header.stamp` (if present) is preserved. The message is re-published under a formatted namespace (e.g., `/rx_robot_{src_id}/...`) so multiple senders remain disambiguated on the Rx robot. Publisher queue depth is configurable. The sink can compute end-to-end latency and delivery statistics from `header.stamp` or wall-clock metadata; these are emitted to standard output

for evaluation.

4.3. Configuration ROS Source (ROS→PDU)

Registry CSV. Path to a CSV mapping topic names to ROS types. One topic per line: `name, type`. The same file (or an identical mapping) must be used on both ends. Up to 16 topics are supported by the header's `Topic ID` space.

Subscribed Topics: Explicit list of topics to subscribe to (Python list, e.g., `['/scan','/odom']`). If left blank, all topics in the registry are subscribed.

Namespace: Source namespace expected on incoming ROS topics (e.g., `robot_1`). Leave blank to match absolute names, or include namespace prefixes directly in the CSV when multiple namespaces are used.

Flush Timeout (s): Maximum wall-clock time to buffer fragments before forcing a packet send. `0.0` disables time-based flushing (size/fit only). Larger values can increase the per-packet entry count but add up latency.

Dest ID: Destination identifier is encoded into the packet header. `255` (`0xFF`) denotes broadcast delivery.

Robot ID: Local sender identifier is encoded into the header (rightmost consecutive digits < 255, e.g., `/robot_34/...` $\rightarrow$ `34`). A value of `-1` enables autodetection from the namespace.

packet_len (bytes): Maximum packet size on the wire, including `Header`, packed `Topic IDs`, `entry` records, `PAD`, and `CRC`. This setting constrains how many entries fit in one packet.

Min. Fragment Size (bytes): Lower bound on the payload bytes per entry. Larger values reduce per-message overhead; smaller values reduce head-of-line blocking for large messages.

Greedy Scheduling *Enabled*: Pack as many entries from the same topic as will fit (higher throughput for large streams). *Disabled*: prefer at most one entry per topic per packet (round-robin–like fairness; lower per-topic latency).

Debug. When enabled, human-readable logs of headers, Topic IDs, and entry layout are transmitted to standard output for inspection.

4.4. Configuration ROS Sink (PDU→ROS)

Registry CSV. Same mapping used by the source, required to de-serialize payloads to the correct ROS message types.

Namespace format. Format string used to republish received topics, with `{src_id}` substituted by the sender ID (e.g., `/rx_robot_{src_id}` $\rightarrow$ `/rx_robot_1/odom`). Ensures separation per-sender at the receiver.

Publisher QoS depth. ROS publisher queue depth for the re-emitted topics. `0` means unbounded. Larger values tolerate bursts; smaller values reduce memory and backpressure latency. The registry must match across endpoints; otherwise, type de-serialization fails. The `Topic ID` supports 16 IDs; reserving one value for control, if desired.

Design constraints. This version does not implement ARQ/FEC, encryption/authentication, routing or duplicate packet suppresion. It is broadcast-only addressing (`dst_id=0xFF`). The design targets functional interoperability across PHYs and clear measurement of latency and packet delivery ratio.

5. Wire Format

Packets have fixed size $P_{\max}$. Zero PAD is inserted *before* the CRC so that the total length is exactly $P_{\max}$.

The packet layout is depicted below and in Fig. 4:

$$\underbrace{\text{Header (5 B)}}_{\text{Version/Flags, Src, Dst, Seq, Hop|EntryCount}}$$

$$\underbrace{\text{Topic IDs } \left(\lceil E/2 \rceil \text{ B}\right)}_{\text{nibbles for } E \text{ entries}}$$

$$\underbrace{\sum_{i=1}^{E} \Big(\text{EntryHdr (5 B)} + \text{Payload } k_i \text{ B}\Big)}_{E \text{ entries}}$$

$$\underbrace{\text{PAD } (0 \ldots 0)}_{\text{covered by CRC}}$$

$$\underbrace{\text{CRC (2 B)}}_{\text{trailer}}$$

5.1. Header and Topic IDs

Topic IDs. Topic identifiers are 4-bit values `0x0,..,0xF`. Consequently, up to 16 application topics may be registered in the CSV. The topic→ID mapping and types must match at both endpoints.

5.2. Entry record and CRC

Each entry carries one entry of a message for the indicated `Topic ID`:

Entry Hdr (5 B): *EntryID* (1 B, random per message), *EntryIndex* (1 B), *EntryTotal* (1 B, total fragments), *PayloadLength* (2 B, big-endian).

Payload (k_i B): entry bytes of the serialized ROS message.

`CRC` is over `Header + Topic IDs + all entries + PAD` with polynomial `0x1021`, init

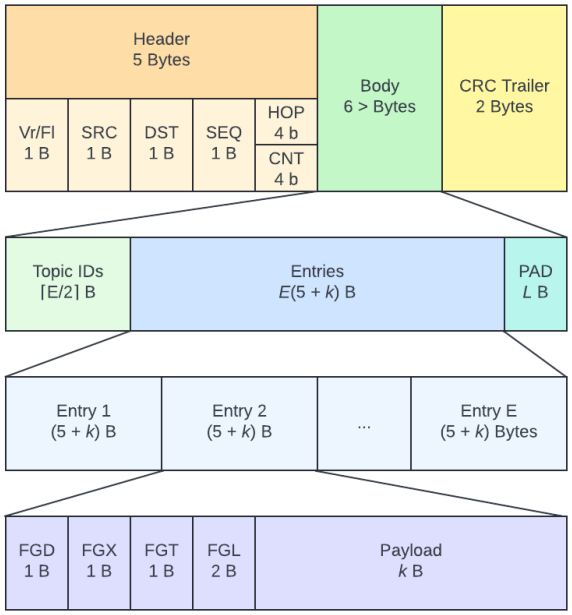

Figure 4: Packet format: Header: *Vr/Fl* (Version/Flags), *SRC* (source), *DST* (destination), *SEQ* (sequence), *HOP* (hop), *CNT* (entry count). Body: *TID* list followed by E *entries*. Each entry has *FGD* (EntryID), *FGX* (EntryIndex), *FGT* (EntryTotal), *FGL* (PayloadLength), and payload. Trailer: *CRC*.

`0x0000`, no reflect, xorout `0x0000`. Non-zero `PAD` is rejected during decoding.

Let E be the number of entries selected for the packet $(1, ..., 15)$. The non-entry overhead is

$$\underbrace{5}_{\text{Header}} + \underbrace{\lceil E/2 \rceil}_{\text{Topic IDs}} + \underbrace{2}_{\text{CRC}} \;=\; 7 + \lceil E/2 \rceil.$$

The byte budget available for all entry records (headers + payloads) is

$$B(P_{\max}, E) \;=\; P_{\max} - \big(7 + \lceil E/2 \rceil\big).$$

Since each entry adds a 5 B entry header, the total payload across the E entries must satisfy

$$\sum_{i=1}^{E} k_i \;\le\; B(P_{\max}, E) - 5E.$$

A convenient upper bound for equal-sized entries is

$$k^\star(E) \;=\; \left\lfloor \frac{B(P_{\max}, E)}{E} \right\rfloor - 5 \;=$$

$$k^\star(E) \;=\; \left\lfloor \frac{P_{\max} - (7 + \lceil E/2 \rceil)}{E} \right\rfloor - 5.$$

Field	Size	Description
Version/Flags	1 B	Bits `[V3..V0|Q1 Q0|C|E]`. Here V=0; $Q \in \{$`00` $= default$, `01` $= latency$, `10` $= bulk\}$; C/E reserved.
Src ID	1 B	Sender identifier (`0,...,255`).
Dest ID	1 B	Destination; `0xFF` denotes broadcast.
Seq Number	1 B	Per-source modulo-256 counter.
Hop|EntryCount	1 B	High nibble: Hop Count (TTL). Low nibble: Entry Count (E).
Topic IDs	$\lceil E/2 \rceil$ B	Packed nibbles for the E entries: even index → high nibble, odd index → low nibble; if E is odd, the final low nibble is 0.

Table 5: Header fields and Topic ID packing.

Special case E=1 (single entry): the maximum payload is $k_{\max}(1) = P_{\max} - 13$ where (13 = 5 Header + 1 Topic ID + 5 frag header + 2 CRC).

Packing policy. When at least two topics are pending, the packetizer targets E=2 and fragments with $k = \max(k^\star(2), k_{\min})$; otherwise, a single-entry packet uses $k = P_{\max} - 13$. Fragments are interleaved *round-robin* across topics. With *Greedy Scheduling* enabled, multiple fragments from the same topic may be packed back-to-back if they fit. A *Flush timeout (s)* may force sending a partially filled packet to limit latency.

Example: With $P_{\max}$=220 B and E=2:

$$B = 220 - (7 + \lceil 2/2 \rceil)$$

$$B = 220 - 8 = 212$$

$$k^\star(2) = \left\lfloor \frac{212}{2} \right\rfloor - 5 = 106 - 5 = 101 \text{ B}.$$

Thus, each entry can carry up to 101 B of payload; the remainder becomes PAD to reach exactly $P_{\max}$.

5.3. PAD Length

Given entry sizes k_i, the PAD length is

$$L_{\text{pad}} = P_{\max} - \left(5 + \lceil E/2 \rceil + \sum_{i=1}^{E} (5 + k_i) + 2\right) \;\ge\; 0,$$

and decoding verifies that all PAD bytes are zero.

5.4. Cardinalities and Limits

`Topic IDs` span `0x0,...,0xF` (up to 16 registered topics). The per-packet `EntryCount` field limits $E \in$

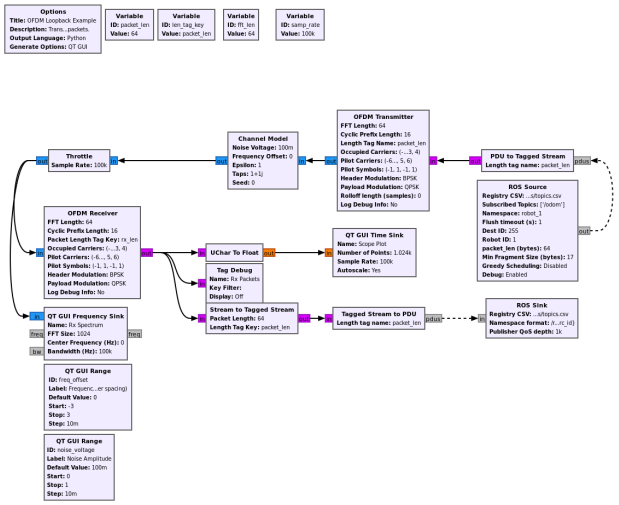

Figure 5: OFDM Tx/Rx loopback flowgraph with ROS source/sink blocks.

$1, ..., 15$. Hop Count $\in 0, ..., 15$. Packets with E=0, negative PAD, or non-zero PAD are rejected. CRC mismatches are dropped.

The following reflects the current `packet.py` implementation: `make_flags()` builds `Version/Flags` $(V$=0, 2-bit QoS policy, reserved compression/encryption bits$)$. `_pack_header()` constructs the 5 B header `(flags, src, dst, seq, hop|count)`. Topic IDs are nibble-packed with odd-E low-nibble padding, entry headers use big-endian `PayloadLength`, PAD is added to meet $P_{\max}$, and `CRC()` computes the trailer.

6. Experimental Setup

We demonstrate our new ROS-GNU Radio blocks in a two-robot simulation: `robot_1` publishes `/odom` at rate R Hz; the *ROS→PDU* block (`ros_rx`) packetizes and injects PDUs into a GNU Radio OFDM loopback Tx-Rx flowgraph (see Fig. 5; the *PDU→ROS* block (`ros_tx`) reconstructs the messages and republishes them under `/rx_robot_{src_id}/odom` (see Section 4). We also test direct (no PHY) loopback in GNU Radio, where the PDU stream is looped back between ROS Source and ROS Sink blocks in GNU Radio with a fixed MTU size of 64 B.

We measure: 1) packet delivery ratio defined as the number of messages delivered to the ROS Sink over the number of messages transmitted from the ROS Source; 2) End-to-end latency, from ROS header stamp at the ROS Source to republish time at the ROS Sink; 3) Path reconstruction error measured by interpolating both paths to a common arc-length grid and compute mean 2D Euclidean distance; this discounts harmless time-stretching due to subsampling.

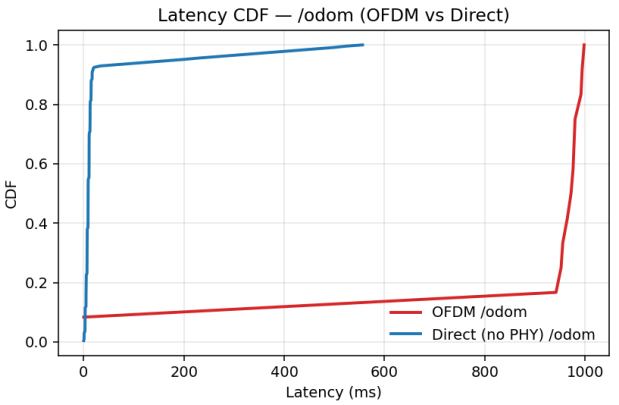

Figure 6: End-to-end latency CDF for `/odom`, comparing OFDM and direct (no PHY) loopback.

Config	Duration [s]	Sent	Recv	PDR	Goodput [B/s]
OFDM	38.0	1328	12	0.01	228.6
Direct	15.0	452	452	1.00	21767.9

Table 6: PDR and goodput for `/odom` at 15 Entries-per-message.

7. Experimental Results

Packet Delivery Ratio and Goodput. Table 6 summarizes total duration, delivered counts, packet delivery ratio (PDR), and application-level goodput (payload bytes/s). Values reflect the ROS Sink's standard output logs accounting over the entire experiment duration.

Latency. Figure 6 depicts end-to-end latency measured for a direct (no PHY) and OFDM loopback in GNU Radio. The direct path exhibits lower and tighter latency, as expected. The OFDM path shows higher latency due to PHY framing and buffering; however, the bulk of packets delivered remain within a limited region that supports coherent trajectory reconstruction (as we observe in Fig. 7).

Path Reconstruction. Figure 7 illustrates the 2D path traced from the transmitted odometry from robot_1 (blue color) and the received odometry of robot_1 at robot_2 (red color). Blue markers correspond to samples generated every 50 Tx messages. Red markers correspond to samples generated for every single received message. In our OFDM configuration, the link goodput is below the source publish rate, producing a sparser trajectory. The ROS Sink block delivers a consistent *subsampled* path with limited latency, but some intermediate poses of robot_1 are dropped. The received odometry path remains geometrically consistent with the source despite subsampling under bandwidth pressure (Figure 7). This indicates functional utility for state sharing even when goodput falls below the publish rate.

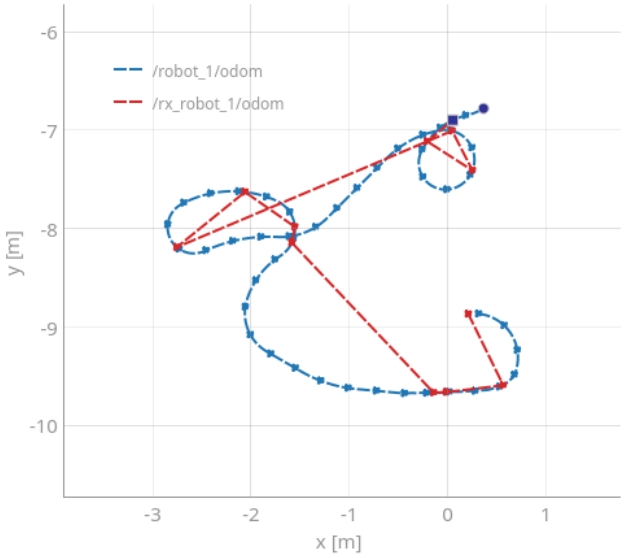

Figure 7: Trajectory representation of `/odom` over loopback OFDM: transmitted `/robot_1/odom` (blue) vs. received `/rx_robot_1/odom` (red). Blue marks every 50 sent messages; red marks every received message. The OFDM link under this configuration yields a sparser but coherent received path.

8. Limitations and Future Work

ROS 2 typically transports messages via DDS over IP. *rosbridge* exposes topics via WebSocket/JSON. IP-over-SDR provides another path but retains the IP stack. Our new GNU Radio blocks instead enable experimentation with custom PHY/MAC/NET protocol stack –that are not necessarily IP-compliant– with ROS semantics preserved. We consider only broadcast-only addressing and the QoS policy is optional. Duplicate suppression and ACK-based retransmission are out-of-scope. Future versions will address ARQ/FEC, encryption, authentication, and routing.

9. Conclusion

We developed a simple, effective link adapter for ROS 2 over GNU Radio that provides a compact wire format, and works across disparate PHY layers. This decoupling enables controlled experiments with custom protocol stacks (non IP-compliant) on SDRs while maintaining ROS types and developer ergonomics.

References

Dds interoperability wire protocol (ddsi-rtps), version 2.3, 2019.

Bloessl, Bastian, Segata, Michele, Sommer, Christoph, and Dressler, Falko. Towards an open source ieee 802.11p stack: A full sdr-based transceiver in gnu radio. In *2013 IEEE Vehicular Networking Conference*, pp. 143–149, 2013. doi: 10.1109/VNC.2013.6737601.

Crick, Christopher, Jay, Graylin, Osentoski, Sarah, Pitzer, Benjamin, and Jenkins, Odest Chadwicke. Rosbridge: Ros for non-ros users. In *Robotics Research*, volume 100 of *Springer Tracts in Advanced Robotics*, pp. 493–504. Springer, 2017. doi: 10.1007/978-3-319-29363-9_28.

Gielis, Jennifer, Shankar, Ajay, and Prorok, Amanda. A critical review of communications in multi-robot systems. *Current Robotics Reports*, 3:349–364, 2022. doi: 10.1007/s43154-022-00090-9.

Gomez-Miguelez, Ismael, Garcia-Saavedra, Andres, Sutton, Paul D., Serrano, Pablo, Cano, Cristina, and Leith, Doug J. srslte: an open-source platform for lte evolution and experimentation. In *Proceedings of the Tenth ACM International Workshop on Wireless Network Testbeds, Experimental Evaluation, and Characterization*, WiNTECH '16, pp. 25–32, New York, NY, USA, 2016. Association for Computing Machinery. ISBN 9781450342520. doi: 10.1145/2980159.2980163. URL https://doi.org/10.1145/2980159.2980163.

Jiao, Xianjun, Liu, Wei, Mehari, Michael Tetemke, Aslam, Muhammad, and Moerman, Ingrid. openwifi: a free and open-source ieee 802.11 sdr implementation on soc. In *2020 IEEE 91st Vehicular Technology Conference (VTC2020-Spring)*, 2020. doi: 10.1109/VTC2020-Spring48590.2020.9128614.

Kang, Woochul, Kapitanova, Krasimira, and Son, Sang Hyuk. Rdds: A real-time data distribution service for cyber-physical systems. *IEEE Transactions on Industrial Informatics*, 8(2):393–405, 2012. doi: 10.1109/TII.2012.2183878.

Macenski, Steve, Foote, Tully, Gerkey, Brian, Lalancette, Chris, and Woodall, William. Robot operating system 2: Design, architecture, and uses in the wild. *Science Robotics*, 7(66), 2022. doi: 10.1126/scirobotics.abm6074.

Morita, Ren and Matsubara, Katsuya. Dynamic binding a proper dds implementation for optimizing inter-node communication in ros2. In *2018 IEEE 24th International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, pp. 246–247, 2018. doi: 10.1109/RTCSA.2018.00043.

Nikaein, Navid, Marina, Mahesh K., Manickam, Saravana, Dawson, Alex, Knopp, Raymond, and Bonnet, Christian. Openairinterface: A flexible platform for 5g research. *SIGCOMM Comput. Commun. Rev.*, 44

(5):33–38, October 2014. ISSN 0146-4833. doi: 10.1145/2677046.2677053. URL https://doi.org/10.1145/2677046.2677053.

Project, Eclipse Zenoh. rmw_zenoh: A ros 2 middleware implementation based on zenoh. https://github.com/ros2/rmw_zenoh, 2025. Accessed: 2025-09-08.

Quigley, Morgan, Conley, Ken, Gerkey, Brian, Faust, Josh, Foote, Tully, Leibs, Jeremy, Wheeler, Rob, and Ng, Andrew Y. Ros: an open-source robot operating system. In *ICRA Workshop on Open Source Software*, 2009.

Sisbot, E. Akin, Vega, Augusto, Paidimarri, Arun, Wellman, John-David, Buyuktosunoglu, Alper, Bose, Pradip, and Trilla, David. Multi-vehicle map fusion using gnu radio. In *Proceedings of the GNU Radio Conference*, 2019. EPOCHS Reference Application (ERA).

Tan, Kun, Zhang, Jiansong, Fang, Ji, Liu, He, Ye, Yusheng, Wang, Shen, Zhang, Yongguang, Wu, Haitao, Wang, Wei, and Voelker, Geoffrey M. Sora: High performance software radio using general purpose multi-core processors. In *USENIX NSDI*, pp. 75–90, Boston, MA, 2009.