Rover Communications

- Overview
- UDP Forwarder and ROS2 Publisher
- Protobuffers
- PBEnvelope
- Integration with ROS2
- Adding a new message
- Testing with udp_client
- Dev Environment Setup
- Known Limitations

Overview

Communication is a subsystem responsible for bridging the two distinct networks present on the rover: the Ethernet Network and the ROS2 Network. Without it, Jonny Boi (the Jetson Orin Nano) would have no way to talk to the hardware microcontrollers or the Basestation, and the internal ROS2 software stack would have no way to act on external inputs or send commands outward.

On the Ethernet side, the Jetson connects to the hardware boards (ArmBoard, DriveBoard, SensorBoard, ContainerBoard) and to the Basestation over WiFi. On the ROS2 side, the Communications node exchanges messages with internal nodes such as the Behavior Node, which handles business logic, and the Controller Node, which manages hardware commands.

All messages sent over the network, whether towards the Basestation or the hardware microcontrollers, are encoded as Protobuffers, wrapped inside a PBEnvelope. The PBEnvelope acts as a typed container that lets the receiver identify what kind of message it is receiving before deserializing the payload. These wrapped messages are transmitted as raw bytes inside UDP packets.

The Communications node operates in both directions:

- Inbound (Ethernet -> ROS2): UDP packets received from the Basestation or hardware boards are parsed, the protobuf payload is extracted from the PBEnvelope, and the data is converted into custom ROS2 messages before being published over the appropriate ROS2 topic for other nodes to consume.
- Outbound (ROS2 -> Ethernet): The Communications node subscribes to relevant ROS2 topics, converts the incoming ROS2 messages back into Protobuffers, wraps them in a PBEnvelope, and transmits them as UDP packets to the intended destination, either the Basestation or a hardware board.

Consider the Rover Architecture diagram Rover Architecture diagram to get a visual overview of how the Communications node sits at the center of these data flows.

Before diving into the codebase, it is strongly recommended to read the ROS2 documentation, particularly the entries regarding topics, messages and publishers/subscribers:

https://docs.ros.org/en/humble/Concepts/Basic/About-Interfaces.html

You will also need background knowledge in C++ and UDP networking. The following guide is a solid starting point:

https://beej.us/guide/bgnet/html/split-wide/

UDP Forwarder and ROS2 Publisher

`udp_forwarder_node.cpp`

This is the main file running the communications logic on Jonny Boi's side. It is a ROS2 node that simultaneously acts as a UDP server and a ROS2 publisher/subscriber. Its primary responsibilities are:

- Receiving raw encoded PBEnvelope packets over UDP from the Basestation or hardware microcontrollers
- Deserializing the protobuf payload and converting it into a custom ROS2 message
- Publishing that ROS2 message over the appropriate topic for internal nodes to consume
- In the outbound direction, subscribing to ROS2 topics and converting messages back into protobufs to be sent as UDP packets

The Handler Pattern

The most important architectural concept in this file is the **registry-based handler dispatch system**. Rather than having a large switch statement that handles every possible message type, the node maintains an `unordered_map` that maps each `PBEnvelope::PayloadCase` enum value to a dedicated handler object.

When a UDP packet arrives, the node parses it as a `PBEnvelope` and calls `payload_case()` to identify the message type. It then looks up the corresponding handler in the map and calls `handle(envelope)` on it. Each handler is a class that extends the abstract `Handler` base class and is solely responsible for one message type — deserializing the protobuf, mapping its fields to a ROS2 message, and publishing it on the correct topic.

Registering a handler looks like this:

```
handlers_.emplace(
    static_cast<int>(PBEnvelope::kImuInfo),
    std::make_unique<ImuHandler>(this, "imu_data", 10)
);
```

The three constructor arguments are the node pointer (so the handler can create a publisher), the ROS2 topic name to publish on, and the publisher queue size. Adding support for a new message type is as simple as writing a new handler class and adding one `emplace` call here — the rest of the dispatch logic requires no changes. See the **Adding a New Message Type** page for a step-by-step guide.

The rx_loop Background Thread

UDP receiving does not happen on the ROS2 spin thread. Instead, the node spawns a dedicated background thread that runs `rx_loop()` continuously, blocking on `recvfrom()` until a datagram arrives. This design ensures that the ROS2 executor is never blocked waiting for network I/O, and that incoming packets are processed as fast as the network delivers them regardless of what the ROS2 stack is doing.

This is worth keeping in mind when debugging. If you see unexpected behavior related to timing or message ordering, the source may be a race condition between the rx_loop thread and the ROS2 spin thread. The `running_` flag is declared as `std::atomic<bool>` specifically to ensure safe cross-thread signaling during shutdown.

UDP Forwarding Behavior

Before dispatching to a handler, the node forwards every raw UDP datagram to both `dstA_` and `dstB_` unconditionally. This means every message type, regardless of its intended destination, is forwarded to both addresses. This is a known architectural limitation — see the **Known Limitations** page for more context and the rationale behind it.

Configuration Parameters

The node's network configuration is fully driven by ROS2 parameters declared at startup:

Parameter	Default	Description
`listen_port`	5000	UDP port the node listens on
`dst_a_port`	6000	First forwarding destination port
`dst_b_port`	6001	Second forwarding destination port
`dst_ip`	127.0.0.1	Destination IP for both forwarding targets

These can be overridden at launch time via a ROS2 launch file or `--ros-args` flags without recompiling, making it easy to reconfigure the node for different network environments such as the

Jetson on the rover versus a local development machine.UDP Forwarding

Currently, every received UDP packet is forwarded raw to both `dstA_` and `dstB_` before handler dispatch.

> This implementation will CHANGE SOON. to forward ONLY to appropriate destinations.

Protobuffers

What are Protobuffers?

Protocol Buffers (Protobuffers or protobufs) are Google's binary serialization format for structured data. Compared to alternatives like JSON or XML, they are significantly more compact and faster to serialize/deserialize, making them well-suited for real-time communication between the rover's subsystems. They are also strongly typed and language-agnostic, the same `.proto` definition can generate code for C++, Python, Rust, and others. For a general introduction, refer to the official documentation:

https://protobuf.dev/overview/

The ERC-Protobufs Repository

All protobuf definitions used by the rover live in a shared repository called **ERC-Protobufs**, which is used as a dependency across the rover's software stack. You can find it linked from the Starting with Protobufs page.

The repository is organized under a `components/` folder, where each subfolder groups messages by the hardware board or subsystem they belong to:

```
components/
  arm_board/
  driving_board/
  sensor_board/
  container_board/
  basestation/
  common/
```

For example, `components/basestation/detected_object.proto` contains the message definition for a single YOLO-detected object sent from Jonny Boi to the Basestation. Placing it under `basestation/` makes its intended direction and purpose immediately clear.

Naming Conventions

Every message name in the repository must be **globally unique**, regardless of which folder it lives in. This is because proto3 imports reference files by path, but message names exist in a flat global namespace, two messages with the same name will cause build failures.

The convention used across the rover is to prefix every message name with its component:

- `ArmBoardControlSignals`
- `BasestationControlMode`
- `SensorBoardIMUInfo`
- `DrivingBoardMotorMessage`

Follow this convention strictly when adding new messages.

How Protobufs are Compiled

The `comms` package does not manually clone ERC-Protobufs. Instead, CMake fetches it automatically at build time using `FetchContent`, pinned to a specific commit hash in `CMakeLists.txt`:

```
FetchContent_Declare(
  erc_protobufs
  GIT_REPOSITORY https://github.com/RoboTeamTwente/ERC-Protobufs.git
  GIT_TAG <commit_hash>
)
```

Once fetched, `protoc` generates `.pb.h` and `.pb.cc` files into the build directory, which are then compiled into the `comms` node. This means that whenever new proto files are added to ERC-Protobufs and committed, the `GIT_TAG` in `CMakeLists.txt` must be updated to the new commit hash before the changes will be picked up by the build. The full procedure is covered in the **Adding a New Message** page.

The PBEnvelope

All protobuf messages on this rover are wrapped inside a `PBEnvelope` before being transmitted over UDP. The PBEnvelope acts as a typed container that allows the receiver to identify which type of message is inside before deserializing the payload. See the dedicated PBEnvelope page for a full explanation.

PBEnvelope

What is the PBEnvelope?

The `PBEnvelope` is a wrapper message defined in `components/common/envelope.proto` in the ERC-Protobufs repository. Every protobuf message sent over UDP on this rover — whether from the Basestation, the Jetson, or a hardware microcontroller — is wrapped inside a `PBEnvelope` before transmission.

When a raw UDP datagram arrives, the receiver has no way of knowing what type of message is inside without some kind of type identifier. The PBEnvelope solves this by using proto3's `oneof` field, which encodes the message type directly into the serialized bytes. The receiver calls `payload_case()` on the deserialized envelope to determine the message type before extracting and deserializing the actual payload.

Structure

The PBEnvelope uses a `oneof payload` block where each field corresponds to one registered message type. Only one field can be set at a time — that is the nature of `oneof`. When serialized, proto3 encodes which field is set, allowing the receiver to call `envelope.payload_case()` and get back an enum value like `PBEnvelope::kImuInfo` or `PBEnvelope::kControlMode`.

Registered Message Types

Below is the full table of currently registered payload cases, their field numbers, and their communication direction:

Field Number	Message Type	Direction
1	SensorBoardPHInfo	Hardware → ROS2
2	ArmBoardControlSignals	ROS2 → Hardware
3	ArmBoardDiagnostics	Hardware → ROS2
4	ArmBoardMovementFeedback	Hardware → ROS2
5	ArmBoardActualPositions	Hardware → ROS2
6	ArmBoardTargetMovement	ROS2 → Hardware
7	ArmBoardObstructions	Hardware → ROS2

8	DrivingBoardDiagnostics	Hardware → ROS2
9	DrivingBoardMotorMessage	ROS2 → Hardware
10	DrivingBoardMotorPeriodicProgress	Hardware → ROS2
11	SensorBoardDiagnostics	Hardware → ROS2
12	SensorBoardGPSInfo	Hardware → ROS2
13	SensorBoardIMUInfo	Hardware → ROS2
14	SensorBoardLoadCellInfo	Hardware → ROS2
15	SensorBoardPressureInfo	Hardware → ROS2
16	BasestationControlMode	BS → ROS2
17	BasestationDetectedObject	ROS2 → BS
18	BasestationObjectSelection	BS → ROS2
19	BasestationRockMeasureRequest	BS → ROS2
20	BasestationRockMeasureResult	ROS2 → BS
21	BasestationRoverLocalization	ROS2 → BS

Important: Field numbers in a `oneof` block are permanent. Once a field number is assigned to a message type, it must never be reused for a different type — even if the original message is removed. This would break deserialization for any system still using an older version of the envelope. Always use the next available field number when adding a new entry.

A note on `BasestationDetectedObject`

Unlike most messages which carry a complete snapshot of state, `BasestationDetectedObject` sends **one detected object per message**. This is because Embedded requires fixed-size protobuf messages and cannot handle `repeated` fields (which are dynamically sized). The Basestation reconstructs the full frame by collecting all messages sharing the same `frame_id` until it has received `total_count` of them. See `components/basestation/detected_object.proto` for the full definition.

A note on `message_types.proto`

> TLDR: I don't remember what we were doing with this file.

The ERC-Protobufs repo contains a file called `message_types.proto` which defines a `PBMessageType` enum. This file is a leftover from an earlier architecture where a manual type ID system was

planned. It is explicitly excluded from the build in `CMakeLists.txt` because its enum values collide with message names in the global proto3 namespace. It is unused, do not reference it or add new entries to it.

Trailing Zero Bytes

The `envelope.proto` file includes an important note: all PBEnvelopes sent over the network must have trailing zero bytes removed before transmission. Proto3 serializes unset fields as zero bytes, and stripping them reduces network traffic. The receiver must pad the received datagram back to the full PBEnvelope size before deserializing. This is handled by the protobuf library's `SerializeToString` and `ParseFromArray` functions when used correctly.

Adding a New Message Type

To add a new message type to the PBEnvelope, see the dedicated [Adding a new message](#) page for the full step-by-step procedure.

Integration with ROS2

Overview

The Communications node bridges the gap between raw UDP/protobuf packets and the ROS2 ecosystem running on Jonny Boi. This page explains how incoming protobuf messages get converted into ROS2 messages and published on topics, the handler pattern that makes this extensible, and how custom ROS2 message types are defined.

Custom ROS2 Message Types

ROS2 uses its own message format (`.msg` files) to define the structure of data exchanged between nodes over topics. These are separate from protobuf definitions, they exist so that internal ROS2 nodes like the Behavior Node can work with typed, idiomatic ROS2 messages rather than raw protobuf objects.

All custom message definitions live in the `msg/` folder of the `comms` package:

```
comms/msg/
  ImuSensorInformation.msg
  SensorBoardGPSInfo.msg
  SensorBoardPHInfo.msg
  SensorBoardDiagnostics.msg
  SensorState.msg
```

These are registered in `CMakeLists.txt` via `rosidl_generate_interfaces`, which generates the corresponding C++ types at build time. The generated types follow the naming convention `comms::msg::MessageName` and can be included in any C++ file as `#include "comms/msg/message_name.hpp"`. For more on ROS2 interfaces, refer to the official documentation:

https://docs.ros.org/en/humble/Concepts/Basic/About-Interfaces.html

The Handler Pattern

The handler pattern is the core architectural decision that makes adding new message types straightforward without ever touching the dispatch logic in `udp_forwarder_node.cpp`.

Every message type that needs to be converted and published as a ROS2 topic is represented by a handler class. All handlers extend the abstract `Handler` base class defined in `include/comms/udp/handler.hpp`:

```
class Handler {
public:
    virtual ~Handler() = default;
    virtual void handle(const PBEnvelope& envelope) = 0;
};
```

Each concrete handler class — one per message type — lives in `include/comms/udp/handlers/` and its corresponding `.cpp` in `src/handlers/`. Its job is exactly three things:

1. Extract the specific protobuf message from the `PBEnvelope`
2. Map the protobuf fields to the equivalent ROS2 message fields
3. Publish the ROS2 message on the appropriate topic

As an example, `ImuHandler` extracts `SensorBoardIMUInfo` from the envelope, maps its accelerometer, gyroscope and magnetometer fields to a `comms::msg::ImuSensorInformation` message, and publishes it on the `imu_data` topic.

Handler Registration

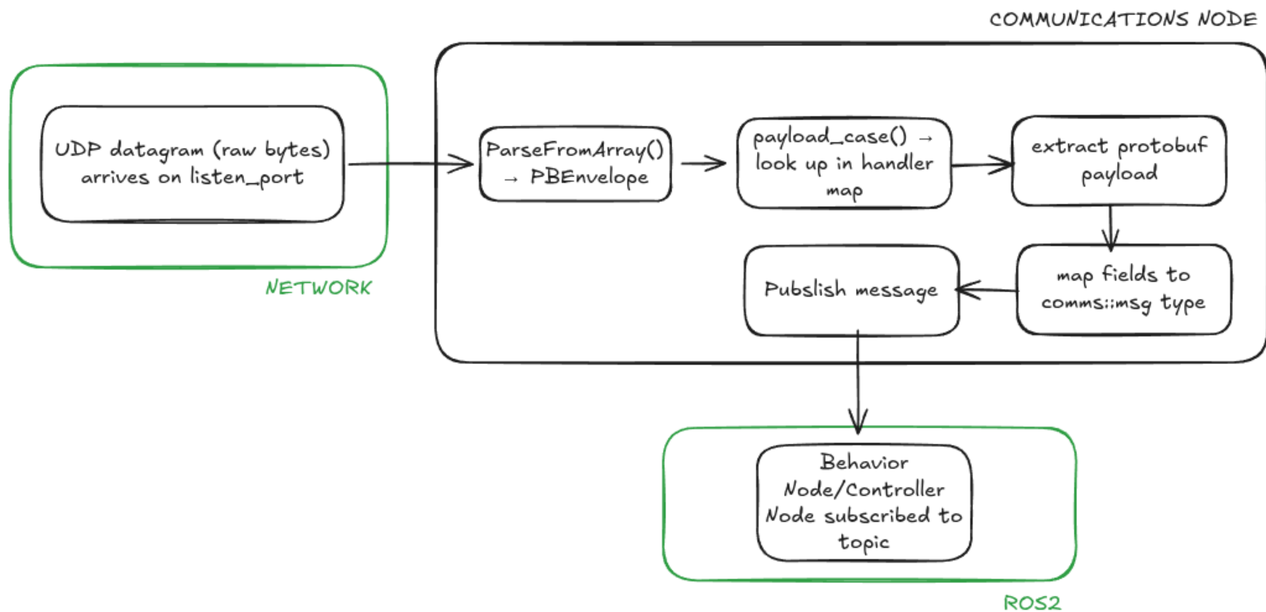
Handlers are registered in the constructor of `udp_forwarder_node.cpp` using a simple `unordered_map` keyed by `PBEnvelope::PayloadCase`:

```
handlers_.emplace(
    static_cast<int>(PBEnvelope::kImuInfo),
    std::make_unique<ImuHandler>(this, "imu_data", 10)
);
```

The three constructor arguments are the node pointer (so the handler can create a ROS2 publisher), the topic name to publish on, and the publisher queue size. To add support for a new message type, you write a new handler class and add one line here. The dispatch logic in `rx_loop` requires no changes at all.

The Full Inbound Flow

The complete path from a raw UDP packet to a ROS2 topic looks like this:



If no handler is registered for a given `payload_case`, the node logs a throttled warning and drops the packet. This means unhandled message types fail silently, keep this in mind when debugging missing data.

The Outbound Flow

The outbound direction works in reverse, the Communications node subscribes to a ROS2 topic, converts the incoming message back into a protobuf, wraps it in a PBEnvelope, and sends it as a UDP packet to the appropriate destination. This path is not yet fully implemented for all message types and will be extended as the Behavior Node develops.

Adding a New Message Type

For a step-by-step guide on adding a new message type end-to-end, from the proto definition all the way to a registered handler — see the dedicated Adding a new message page.

Adding a new message

Testing with udp_client

Dev Environment Setup

Known Limitations