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.

Revision #9
Created 2026-04-16 14:13:16 UTC by Andrei Badea
Updated 2026-04-17 13:48:03 UTC by Andrei Badea