1 Design

Our network supports routed messages between any two nodes.
The routing is implemented within the `router` task and the `rx_task`. The `rx_task` is responsible for always listening and adding received packets to a queue without fully parsing them. The `router` task waits until this queue get an element and processes the packets.
The API into our router consists of

- `send_message(recipient, payload, length)` function, and
- `receive_messages(sender, payload, length)` callback.

The `send_message` is designed to be invoked from tasks other than the `router`, and enqueues the messages onto a queue, which is asynchronously processed by the router task. This is important because the router is not allowed to synchronously wait for acks.
The routing mechanism is routing tables on each node with next hop information for each other node in the network. This is not scalable, but sufficient for small networks. The routing tables are computed centrally, by the gateway node, using network discovery mechanism described in the following section, and distributed to each node in the network.

1.1 Network Discovery and Routing Table Calculation

Periodically, the gateway node performs a discovery procedure. It broadcast a discovery request packet. Each node which receives a discovery packet

1. takes note where the source node it came from: this is the next hop to the gateway
2. marks itself as visited
3. responds to the request by sending a response packet to the node from which the request came
4. broadcasts the same discovery request packet

When a node gets a response packet, it forwards it to the next hop towards the gateway by using the hop information it saved in the first step above.
Each discovery procedure is marked by a version sequence number and all of the above actions and checks apply to one sequence number only.
As the gateway receives the responses, it fills a graph data structure (adjacency list). It does so by reading the traversed links from the `path` field, which is stored in the packet. The `path` field contains a trace of the packet's hops over nodes, which is maintained by our lowest-level packet code.
Once it has the graph, the gateway calculates the shortest paths from each node to all other nodes using repeated invocations of Dijkstra's algorithm. From the shortest path information, the gateway node builds the routing tables for all nodes and distributes them.

2 Metrics

2.1 Low Latency

Latency is relevant for the virtual fence in order to detect a fence crossing and notify the safety monitor before the misbehaving robot goes too far out of the fenced region.
Our design is optimized for low latency thanks to the routing by next hop tables, which contain the next hop calculating according to the shortest path to destination by hop count. See Section **??** for info on computing the shortest-path routing tables.

2.2 Reliability

Reliability is important for a virtual fence because the fence crossing event delivery must happen even if some intermediate nodes malfunctioned.
Our design is optimized for reliability by having the routed packets acknowledged and by having nodes update their routing tables when an acknowledgement does not arrive in time. As a result, if a node X dies, the nodes which list X as the next hop in their routing tables, will replace it automatically.
For example, if a node A gets a message to C, and A's routing table says that the next hop to C is B, then A sends the packet to B, but B is dead. If A does not get an ack from B for this packet, then A updates its routing table to no longer list B as the next hop for C. Instead, A picks a neighbor and adds it as the next hop. On next packet, A will not forward it to B but to a different neighbor. Currently, the choice of neighbor is random (except for packet source), but in the future it could be based on last heard time and RSSI, both of which are tracked.