Summary of PathPlannerPMC node

Overview

PathPlanPMCNode is the central class for managing and coordinating a set of "xbots" in a planar motion control system. It handles communication with the hardware (via **XBotCommands** and **SystemCommands**), manages MQTT messaging for distributed control, and implements path planning and execution logic.

Key Responsibilities

1. **Hardware Communication**
    - Uses **XBotCommands** and **SystemCommands** to send commands and query status from the xbots and the PMC (Planar Motion Controller).
    - Manages connection via connection_handler.
2. **MQTT Messaging**
    - Publishes and subscribes to MQTT topics for real-time communication with other system components.
    - Handles incoming messages (commands, subcommands, station setup) and dispatches them to appropriate handlers.
3. **State and Data Management**
    - Maintains dictionaries to track xbot states, positions, commands, tasks, trajectories, and station information.
    - Uses these to synchronize the logical state of the system with the physical state of the xbots.
4. **Path Planning**
    - Uses a Pathfinding class to compute collision-free trajectories for each xbot from their current to target positions.
    - Handles concurrent path planning tasks and ensures only one planner runs at a time.
5. **Command and Trajectory Execution**
    - Executes movement, levitation, landing, and rotation commands for each xbot.
    - Manages cancellation tokens to safely interrupt and restart tasks as needed.
    - Publishes position and state updates at regular intervals.

Main Flow

Initialization

·On construction, connects to the PMC and gains mastership.

·Initializes MQTT publisher and subscriber, sets up topic handlers, and starts background tasks for MQTT and path planning.

·Publishes the list of available xbot IDs.

MQTT Message Handling

- Subscribes to relevant topics (commands, subcommands, station setup).
- On message receipt, dispatches to handlers:
    - **HandleStationSetup**: Updates station and coordinate dictionaries.
    - **HandleCMD**: Updates the current command for an xbot.
    - **HandleSubCMD**: Handles sub-tasks (Levitate, Land, Rotation, or movement to a station).

Path Planning

- When a movement command is received, updates the xBotID_From_To list with the xbot's current and target positions.
- Calls ExecutePathPlanner, which:
    - Cancels any running planner.
    - Starts a new planning task that computes trajectories for all xbots using the Pathfinding class.
    - Publishes the computed trajectories via MQTT.

Trajectory Execution

·ExecuteTrajectory starts a task for each xbot to execute its trajectory step-by-step.

·Issues motion commands to the hardware, waits for completion, and handles buffer management.

·Monitors for cancellation and cleans up as needed.

State Publishing

·PublishPostionsINFAsync runs in a background thread, periodically publishing each xbot's position and state to MQTT topics.

Command Execution

·Handles special commands like Levitate, Land, and Rotation with dedicated methods.

·For movement, waits until the xbot reaches the target, with retry logic if it gets stuck.

Notable Implementation Details

·**Thread Safety**: Uses locks and cancellation tokens to manage concurrency and avoid race conditions.

·**Resilience**: Includes retry logic and error handling for hardware and communication failures.

·**Extensibility**: Topic handlers and message payloads are structured for easy extension.

Summary

PathPlanPMCNode acts as a bridge between the physical xbots, the path planning logic, and the distributed control system via MQTT. It ensures that commands are received, planned, and executed reliably,

Path finding Summary

---

1. **Core Data Structures**

node

Represents a cell in the grid:

·**x**, **y**: Coordinates.

·parent: For path reconstruction.

·**g**, **h**, **f**: Cost values for A* (g = cost from start, h = heuristic to goal, f = g + h).

·obstacle: If true, this cell is not walkable.

·occupied: List of bot IDs currently occupying this cell.

·proximityCost: Penalty for being near obstacles.

grid

Represents the map:

·width, height: Grid dimensions.

·cells: 2D array of node objects.

·**Initialization**: Sets up the grid, marks obstacle zones, and calculates proximity penalties for cells near obstacles.

---

2. *A Pathfinding (aStar method)**

6. **Purpose**: Finds the shortest path from a start to an end node, avoiding obstacles and occupied cells.
7. **Process**:
    - Resets all node costs and parents.
    - Adds the start node to the open list.
    - While the open list is not empty:
        1. Picks the node with the lowest **f** (and lowest **h** as a tiebreaker).
        2. If the end node is reached, reconstructs the path by following parent links.
        3. For each neighbor:
            1. Skips if not walkable or already closed.

2. Calculates movement cost (including proximity penalty).
3. Updates neighbor if a better path is found.

8. **Returns**: The path as a list of nodes and the total cost.

---

3. **Multi-Agent Path Planning**

priorityPlanner

9. **Purpose**: Plans paths for multiple bots, considering all possible priority orders.
10. **Process**:
    - Generates all permutations of bot priorities.
    - For each priority order, calls conflictHandler to try to find non-conflicting paths.
    - Returns the first valid set of paths found.

conflictHandler

11. **Purpose**: Resolves conflicts between bots' paths.
12. **Process**:
    - For each bot (in priority order), computes its path.
    - Uses conflictSearcher to detect conflicts (collisions or close approaches).
    - If a conflict is found:
        1. The lower-priority bot can either wait (stay in place for a timestep) or try to replan around the conflict.
        2. The process repeats until all conflicts are resolved or a maximum number of iterations is reached.

conflictSearcher

13. **Purpose**: Detects conflicts between bots' paths at each timestep.
14. **Process**:
    - For each pair of bots, compares their positions at each timestep.
    - If they are too close (within a threshold), records a conflict.

---

4. **Grid and Unit Conversion**

·**unit_M_To_CM / unit_CM_To_M**: Convert between meters and centimeters for path planning and output.

·**removeOccupied / makeUnWalkable**: Manage which cells are temporarily blocked by bots.

---

5. **Distance and Utility Functions**

·**getDistance**: Calculates Manhattan or Euclidean distance between nodes.

·**getPriorityMatrix**: Generates all possible priority permutations for bots.

·**SegmentToSegmentXYDistance**: Calculates the minimum x/y distance between two movement segments (for precise conflict detection).

---

6. **Usage Flow**

- **pathPlanRunner** is the main entry point:
    1. Converts input positions to grid units.
    2. Checks for endpoint collisions.
    3. Calls priorityPlanner to get non-conflicting paths for all bots.
    4. Converts the result back to the required units.

Summary

·The class supports multi-agent pathfinding on a grid with obstacles and proximity penalties.

·It uses A* for individual pathfinding and a priority-based approach to resolve multi-agent conflicts.

·It iterates through possible priority orders and uses waiting or replanning to resolve conflicts.

If you need a deeper dive into any specific method or logic, let me know!