The backend is divided in 3 layers:

1. Layer1: Mapping
2. Layer2: Fleet Manager
3. Layer3: The Real Time Robot Driver

Let's start by defining the problem.

Problem definition

We must design the central "brain" for a dynamic Multi-Robot Task Allocation and Navigation System for a logistics warehouse.

The goal is to assign and execute a continuous stream of tasks (packet pickups and dropoffs) to a fleet of robots in the most efficient way possible, minimizing overall completion time and robot-idle time.

This must be achieved while respecting a set of complex, real-time constraints:

1. **Dynamic Environment:** The warehouse map has permanent obstacles (walls) but is also subject to dynamic obstacles (e.g., "dropped boxes") that can appear and block paths at any time.
2. **Multi-Robot Deconfliction:** Robots must navigate the same space and **actively avoid collisions** with each other.
3. **Task Constraints:** Robots have a limited **capacity** (1 packet at a time) and **battery life**, requiring them to periodically stop for charging.
4. **Task Allocation (VRP):** The system must intelligently decide **which robot** gets **which task** (or sequence of tasks) from a central queue to be globally efficient.
5. **Dynamic Tasks:** New tasks are continuously added to the queue, requiring the system to **re-plan** and adapt its assignments without stopping.
6. **Centralized Hardware:** The entire "brain" (both the high-level planner and the real-time pathfinder) runs on a **single centralized server**, which creates a computational bottleneck that must be managed.

PROF

Our Solution

To solve this problem, we have proposed a multi-layered architecture that divides and conquers. Each layer has a distinct responsibility:

- **Layer 1 (Mapping):** Provides two "views" of the world: a fast, static map for high-level planning and a detailed, dynamic map for real-time navigation.
- **Layer 2 (Fleet Manager):** The "Strategic Brain." It decides *which robot* should do *which tasks* and in *what order* (solves the VRP).
- **Layer 3 (Robot Driver):** The "Tactical Brain." It executes one command at a time, finding the *actual path* and avoiding all collisions (other robots and dropped boxes).

Layer 1: Mapping

This layer is responsible for translating the static warehouse blueprint into the two map representations required by Layer 2 and Layer 3.

Input

- `Static 2D Bitmap`: A bitmap representing all *permanent* obstacles (walls, shelves, forbidden zones).

Outputs (Generated Maps)

This layer generates and maintains two distinct maps:

1. The Dynamic Occupancy Grid (Live Map)

`Dynamic 2D Bitmap`
This map is the "source of truth" for real-time navigation.

- **What it is:** A fine-grained grid (a copy of the static bitmap) that is **continuously updated** in real-time with all *temporary* obstacles (e.g., "dropped boxes").
- **Purpose:** To provide a 100% accurate, live view of the world for low-level navigation.
- **Consumed By: Layer 3 (The Robot Driver)**. The Theta* pathfinder runs on this grid, and the ORCA loop uses it to "see" static obstacles.

2. The Static Navigation Mesh (Planning Map)

`Static PolygonGraph NavMesh`
This map is the "fast lookup" for strategic planning.

- **What it is:** A graph of large, convex polygons, generated **offline** from *only* the `Static 2D Bitmap`.
- **Purpose:** To drastically reduce the map's complexity. Instead of millions of grid cells, the planner sees a graph of a few hundred polygons. This allows the Fleet Manager to get path costs (e.g., `cost(A, B)`) almost instantly.
- **Consumed By: Layer 2 (The Fleet Manager)**. The VRP solver queries this map thousands of times to build its cost matrix and find the most efficient task assignments.

PROF

Layer 2: The Fleet Manager (IGNORES Temporary Obstacles)

This layer essentially solves 2 NP-Hard problems:

- `Partition Scheduling:` Which robot gets which set of tasks?
- `TSP:` For the set of tasks assigned to each robot, which is the best route itinerary?

Which, for instance, exists a problem named `VRP` **Vehicle Routing Problem** which solves this problem. In fact, our variant is one that combines both:

- `Capacitated VRP:` where each robot can only pick up 1 packet.
- `Dynamic VRP:` packets keep arriving.

Inputs

- `StaticPolygonGraph NavMesh`: (From Layer 1) Used to query all travel costs.
- `Set of Tasks`: A real-time stream of new tasks (pickups, dropoffs) to be assigned.
- `Robot State Data`: A live feed of each robot's status (e.g., `position`, `battery_level`...).

Processing (The "How")

This layer's primary job is to solve our variant of the **VRP**. It does this in a continuous, dynamic loop.

1. **Cost Matrix Generation:** `OFFLINE`
    - It runs the *A algorithm** on the NavMesh to get the *true, optimal travel cost* (time or distance) between any two points (e.g., `cost(P5, D11)`). It runs A* N * N times to get all the weights to go between 2 PoIs (Points of Interest). Each N is a PoI such as Charger Nodes, PickUp Nodes, DropOff Nodes...
    - NavMesh is small, with V polygons and E edges. One A* is O(E + VlogV).
      `Total Cost = O(N^2 * (E + VlogV))`
2. **Add current needed rows to the matrix:** `ONLINE`
    - Just need to add the rows from robot current positions to task PickUp positions.
3. **VRP Solver (Task Assignment):** `ONLINE`
    - The **Cost Matrix**, along with all `Task Constraints` (capacity, battery) and `Robot State Data`, is fed into a **VRP solver** (e.g., a metaheuristic like Tabu Search or HC).

Outputs (Artifacts)

- `Robot Itineraries`: An ordered list of goals (e.g., `[P5, D11, C1]`) sent to each robot. Note how instead of assigning the tasks, it takes a further step and directly gives the nodes that the robot has to visit, to ease Layer3 complexity.
  This is the sole output of this layer.

PROF

Layer 3: The Robot Driver (HANDLES Temporary Obstacles)

This layer is the one that must handle obstacles. The robot looks at its itnierary (e.g. Go to P5). It only cares about getting there as fast as possible without colliding.

This layer will be working by a service and a fast loop executed every, say 50ms.

Inputs

- `Robot Itinerary`: (From Layer 2) The list of goals to complete.
- `Dynamic 2D Bitmap`: (From Layer 1) The live-updated map used for all pathfinding.
- `Live Robot States`: Positions and velocities of *other* robots, needed for collision avoidance.

Processing (The "How")

1. Service (Theta*)

This is the planner that finds the optimal *route* to a goal using the Theta* algorithm on the `Dynamic 2D Bitmap`. It is better than A* because finds smooth, "any-angle" paths.

How it works-First Approach?

1. A priority queue.
   * Priority by heuristic (shortest job first): When a request comes in, we do a *very* cheap calculation: `distance = sqrt((x2-x1)^2 + (y2-y1)^2)`.
   * Priority by Job Type: A robot that is *blocked* and needs a re-plan (`IsReplanNeeded = true`) is **HIGH priority**. A robot that just finished a task and needs its *next* goal is **NORMAL priority**.
2. A thread pool. Let's assign one worker per CPU and let them run in parallel.

Note:

1. A second approach (later), may be running Lazy Theta* instead of normal Theta* to do less operations.
2. A third, more advanced approach, would be dividing the grid into, say 10x10 sectors (like NavMesh). We pre-calculate all the paths *between* the "doors" of these sectors. Now a path request from A to B is now much faster. - The service just finds:
   1. A path from `A` to the "door" of its sector.
   2. A super-fast, pre-calculated path *between sectors*.
   3. A path from the final "door" to `B`.
      This is *much* more complex to set up, but it's the fastest method for very large maps.

When it runs?

This expensive operation is *only* triggered **on-demmand** in any of these cases:
1. The robot gets a **new goal** from its itinerary.
2. A **"proactive pre-plan"** is triggered (e.g., the robot is 10s from its goal, so it pre-calculates the *next* path).
3. The robot's "Smart Check" confirms its current path is **completely** blocked by a new *static* obstacle.

PROF

Problem: What if `Dynamic 2D BitMap` changes while the robot is following the original Theta* path? ORCA handles it trivially adjusting speed and angle.

2. Master Loop (ORCA)

This is the "dumb" driver that runs in a continuous, fast loop (e.g., every 50ms) for *every* robot.

- **What it is:** The **ORCA (Optimal Reciprocal Collision Avoidance)** algorithm.
- **Purpose:** To handle *imminent, local* dangers. The Theta* path (`CurrentPath`) is the *input* that tells the ORCA loop *what to do*.
- **Logic:**
  1. Gets its "Preferred Velocity" from its `CurrentPath`.
  2. Scans its immediate "danger bubble" (e.g., 5-meter radius).

3. Feeds **all** obstacles in this bubble (both *other robots* and *static obstacles* like "dropped boxes") into the ORCA algorithm.
4. ORCA computes the *one* "Safe Velocity" that avoids everything.

```
// This entire function runs every 50ms
void FastLoopManager() {

    // Get all robot data at once
    List<Robot> robots = Get_All_Robot_States();
    // Iterate through every robot
    for (Robot robot : robots) {
        // 1. Get Preferred
        Velocity vec2 preferred_vel = GetPreferredVelocity(robot,
robot.CurrentPath);

        // 2. Sense Nearby Obstacles (robots AND boxes)
        List<Obstacle> local_obstacles = GetLocalObstacles(robot,
robots);

        // 3. Run ORCA
        vec2 safe_velocity = Run_ORCA(preferred_vel, local_obstacles);

        // 4. Send Command to Simulator
        Send_Simulator_Command(robot, safe_velocity);
        }
}
```

Outputs (Artifacts)

- `Safe Velocity Vector`: The final, low-level command (e.g., "set velocity to 0.8 m/s at 32 degrees") sent to the simulator hardware for the next 50ms.

Note: In Physics and Robotics a "velocity" is a vector that includes both speed and angle (direction).

PROF

How this works IRL?

Let `T` be the initial number of Tasks waiting for the Boot-Up (8:00 AM). Let `R` be the total number of Robots in the fleet. Let `Tmax` be the tunable threshold for triggering a full re-plan.

There are only 3 possible Scenarios.

`Scenario A:` The 8:00 AM "Boot-Up"

This is the initial **"Full VRP-Solve"** when the system starts with a large queue of `T` tasks and all `R` robots are idle.

- **Layer 2:**

1. The solver runs *A on the NavMesh** to get all `R` "robot-to-task" costs and adds these rows to the pre-computed `OFFLINE` matrix.
2. It solves the **heavy VRP** for all `T` tasks, generating the optimal initial itineraries for the fleet.
3. It outputs these itineraries to the assigned robots.

- **Layer 3:**

1. This creates the **"Pathfinding Spike"** as all `R` robots (or all *assigned* robots) request a Theta* path from the server at the same time.
2. **How it's handled:** This is not a problem. The **`Theta* Service`** handles this with its internal **Pathfinding Queue**. All `R` requests are added to this queue.
3. The service's **Thread Pool** (e.g., 8 "worker" threads) grabs the first 8 requests and processes them in parallel.
4. The server's *single* **`FastLoopManager`** **(Master Loop)** starts its 50ms loop. It iterates through all `R` robots. At first, all robots have `CurrentPath = null`, so their `PreferredVelocity` is `(0,0)`, and they "actively stop."
5. As the `Theta* Service` workers finish paths and send them to robots (e.g., `R1` gets its path 1.5s later), the `FastLoopManager` loop will start seeing a valid `CurrentPath` for those robots. They will begin to move, one by one. This **staggers the load** and prevents a system crash.

`Scenario B:` "Streaming" New Tasks (Online)

This occurs when new tasks arrive in a **small trickle** (e.g., 1 or 2 at a time).

- **Trigger:** A new batch of tasks arrives where `NewTasks <= Tmax`.

- **Layer 2:**

1. A **full re-plan is NOT triggered.**
2. The server uses a **"Cheap Insertion Heuristic."**
3. It calculates the "cheapest" place to insert each new task into the *existing* itineraries of the robots (e.g., "What is the *extra* time if I add `Task_101` after `R1`'s current job?").
4. It finds the "cheapest" slot (e.g., adding it to `R4`'s queue is the lowest cost) and updates `R4`'s itinerary.

- **Layer 3:**

1. **No immediate effect.** `R4` is not interrupted.
2. When `R4` finishes its current goal, it will simply receive the *newly inserted task* as its next goal and request a Theta* path for it. Obviously it will ask for the path a couple of seconds before arriving to the path so it doesn't stay idle at any time.

`Scenario C:` "Batch" New Tasks (Online)

This occurs when a large set of new tasks arrives at once, making the old plan inefficient.

PROF

- **Trigger:** A new batch of tasks arrives where `NewTasks > Tmax`.
- **Layer 2:**
    1. This triggers the **"Smart, Self-Aware Re-plan."**
    2. The server estimates its own re-plan time (e.g., `T_replan = 45s`).
    3. It starts the **heavy VRP solver** (on *all* unstarted tasks) in the background.
    4. If any robot (`R5`) finishes its job *during* these 45s, the server checks its *next* (old) task's duration (`T_next`).
    5. If `T_next < T_replan` (e.g., a 20s task), the server tells `R5` to **wait**.
    6. If `T_next > T_replan` (e.g., a 90s task), the server tells `R5` to **proceed** (to keep it busy).
- **Layer 3:**
    1. After approximately 45s, the *new, optimal* itineraries are broadcast to all robots, overwriting the old ones.
    2. This may cause a **"mini-spike."** Any robot on a now-canceled path will immediately discard its `CurrentPath` and request a new Theta* path for its new goal.
    3. The **`Pathfinding Queue`** in the `Theta* Service` handles this small spike just as it did in Scenario A.

Cases to see

1. Obstacles to appear randomly (`random_temporary_obstacle_generator`).
2. If 2 robots are trying to go in the same narrow hall (one left and one right), they would stay idle because of ORCA, handle that.
3. Handle pointles theta* re-calculations: Imagine a oibstacle appears and disappears before it affects the robot (it should still maintain the best asisgnment just in case).