Table of Contents

1. Project Title, Authors, and Stakeholder

a. Project Title: ***Life-Long MAPF Distributed System***

b. Team/Project number: 26

c. Team name: MAPFer

d. Team members:

Dongming Shen, USCID: 7170243072

Tiancheng Xu, USCID: 1522206865

Yi Jiang, USCID 5261852930

e. Stakeholder: Christopher Leet

f. TA: Marcelo Laser

2. Preface

This document provides a detailed process report (Until Oct. 30, 2022) for the Life-Long Distributed MAPF (multi-agent path finding) System project.

3. Introduction

Recent years have seen multi-robot systems increase in size and ubiquity. Warehouse operators such as Amazon routinely service large warehouse complexes with teams of thousands of robots. Entertainment companies such as Spaxels produce aerial spectaculars consisting of thousands of drones. Each of these systems must solve the Multi-Agent Path Finding (MAPF) problem, the problem of finding collision-free paths for a group of agents operating in a known common environment. A practical solver must:

(a) Produce near-optimal paths

(b) Solve very large (multi-thousand robots) MAPFinstances in real-time

(c) Be capable of adjusting its solution if a robot completes its goal and is assigned a new goal, and

(d) Be capable of adjusting its solution if a robot's movement fails.

To date, industrial multi-robot systems solve the MAPF problem using a traffic system-based approach. In a traffic system-based approach, the environment that agents operate in is divided into components such as roads and intersections, much like a city traffic system. Simple local rules govern the behavior of agents in each component. This approach is scalable and robust to new goal assignment and movement failure but is often far from optimal.

In recent years, academia has seen the development of search-based approaches to MAPF, such as conflict-based search. These approaches can be made optimal/bounded suboptimal but fail to scale and struggle to handle movement failure and goal reassignment.

A hybrid approach promises to combine the best of both systems. In this approach, open spaces controlled by a search-based MAPF algorithm are connected using a traffic-system-based algorithm. Each search-based MAPF instance and components of connecting traffic system are run in a separate thread, providing a high degree of scalability. Only one component must replan should an agent's movement fail or goal change, providing robustness to these conditions.

This project involves key ideas from distributed systems, rule-based AI, and search-based AI hot topics in modern industry in the context of a real research problem with direct application to industry. The main goal of this project is to build a system that can solve MAPF problems involving over ~10000 agents with acceptable efficiency.

4. Important Terms Definition & Introduction

Highway:

The road system. Multiple highways will connect the towns. The agent needs to go through these highways to go from one Town to another. It will take agents who just leave a town and then plan the paths for them to arrive at their destination town where their goals are. Use a rule-based MAPF algorithm.

Highway Node:

Each grid/node in this highway system (these nodes compose the highways). Each of these nodes should either have exactly one out-going direction (w, e, n, s) or be an intersection (multiple out-going directions).

Town:

The map is split into different regions known as towns. Each Town runs independently to direct the agents to their goals. Each Town takes the new agents from the Highway through locks, runs EECBS to plan the paths for agents in it, and sends agents to the Highway with locks. Agents stay around in the Town while the Highway is crowded.

Town Node:

Each grid/node in a town. Town grid can be either lock('L'), obstacle('@'), or empty cell('.'). Each node is connected to another one in four directions: up, down, left, and right.

Lock Node:

Serves the buffer node between Town & Highway to avoid bottleneck crowding agents that negatively affect the system's throughput. For example, if many agents want to go from Town_1 to Town_2 through Highway simultaneously while Highway is narrow, then in Chris's previous system, agents might get stuck in Highway and, therefore, also stuck around at Town_1 to Highway's entrance. However, adding the "Lock" concept, most agents in Town_1 will not even get close to Highway's entrance with a high probability. This is because before an agent in Town_1 is assigned a Lock_out (to enter Highway from Town_1) and a Lock_in (to enter Town_2 from Highway), it will wander around in Town_1. In this way, an agent in Town_1 will not get stuck around the Highway entrance but will wait in other places until it gets "called" and can then go to Town_2 more efficiently with a much lower probability of getting stuck. This idea aims to improve the system's throughput.

Global Coordinator:

The Global Coordinator serves to 1. Get new tasks (goals) for the free agents in Town, and assign Locks (if available) to the agents, or set the agent's property so that it will wander around in its Town; 2. Handle messages (thread communications) between the Town system and the Highway system; 3. Free locks when an agent finishes using it.

Agent:

Each agent has a current_location and a goal_location (parsed by the Global Coordinator from Tasks). Usually, current_location and goal_location are in different towns (for example, Town_1 and Town_2). Then after assigning the agent's goal_location, the Global Coordinator should also check if there is a (Lock_out (in Town_1), Lock_in (in Town_2)) pair, where both Locks are free and connected by Highway. If it exists, the Global Coordinator should assign Lock_out and Lock_in to the agent and lock these Locks to prevent conflicts. If it does not exist, the Global Coordinator should assign the agent a random goal in Town_1 to make the agent wander around. In rare cases, if current_location and goal_location are in the same Town, then the Global Coordinator should not worry about the Locks but will simply assign goal_location to the agent for the Town to route the agent to its goal directly.

Routing Agent:

(We will focus on the general case where an agent has current_location in Town_1, goal_location in Town_2, and is assigned Lock_out in Town_1 and Lock_in in Town_2. This is because other cases are only sub-problems of this general case.) H

ere are the steps to route the agent from its current_location to its goal_location:

1. Town_1 will route the agent from its current_location to any empty adjacent node Adj_Lock_1 of Lock_1 in Town_1.
2. Upon arriving at the Adj_Lock_1, Town_1 tells Global this information; Global then tells Highway this information; Highway blindly transfers the agent from Adj_Lock_1 to Lock_1 (which should be empty and is therefore safe).
3. Highway will route the agent from Lock_1 to Lock_2. Upon leaving Lock_1, Highway will tell Global this information; then Global will free Lock_1.

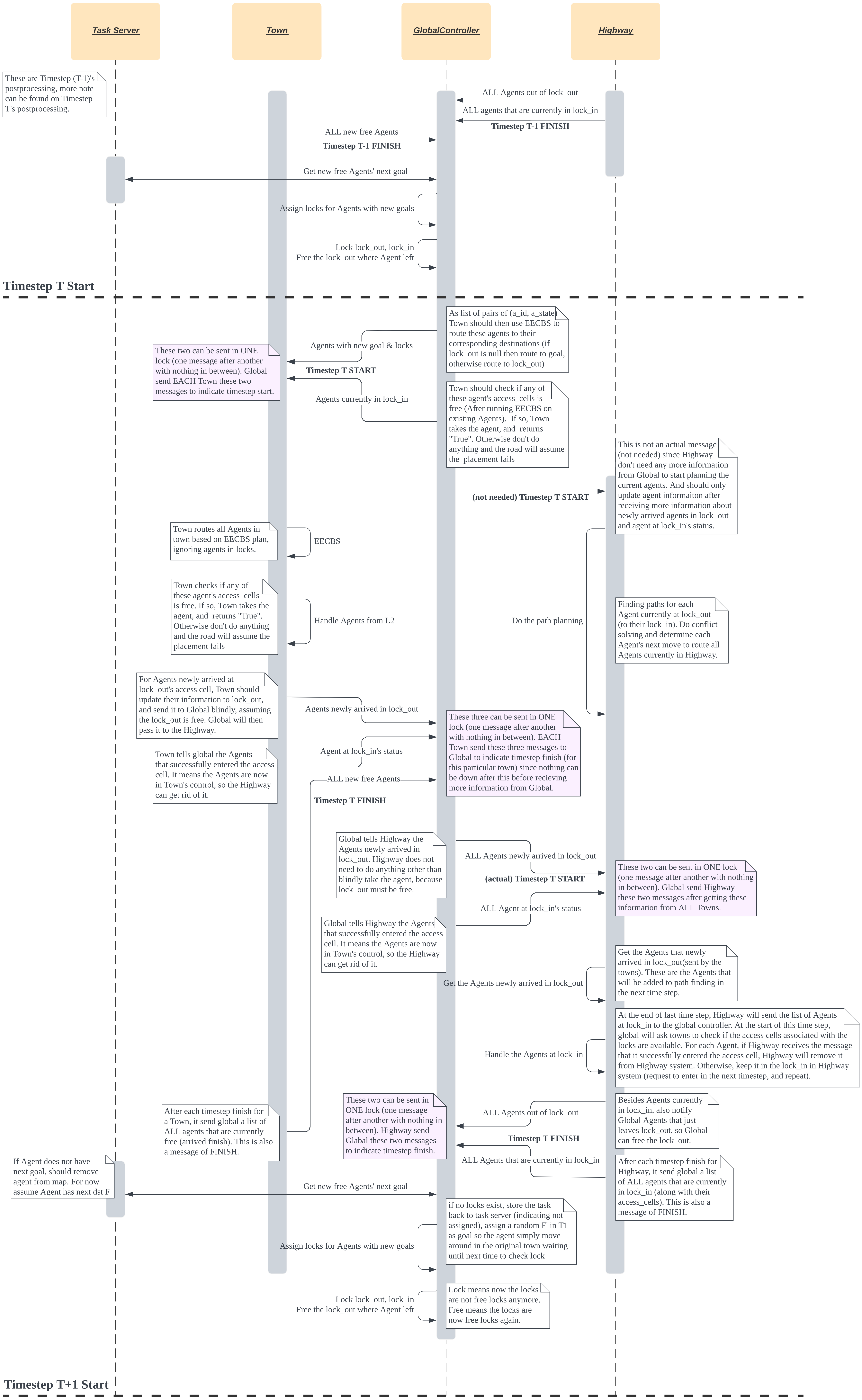
4. Upon arriving at Lock_2, Highway tells Global this information, Global then tells Town_2 this information. Town_2 checks if any adjacent node of Lock_2 in Town_2 is empty: if so, it transfers the agent from Lock_2 to this empty Adj_Lock_2 and tells Global the information; Global then tells Highway this information, so Highway will remove the agent from Lock_2; Global will also free Lock_2; if not, nothing happens, the agent will stay in Lock_2 and keep requesting every timestep until getting transferred into Town_2.
5. Finally, Town_2 will route the agent from Adj_Lock_2 to goal_location. When it arrives, Town_2 tells Global that the agent has reached the goal and is now free.

Message:

Message is used for thread communications. It contains the actual data to be sent from one component to another. For example, by the end of a timestep, each Town should construct a message that includes all new free agents that just arrived at the goal and send the message to Global. Then, the global Coordinator constructs and sends the message that assigns the goals for new free agents back to Town. The detailed message types are included in the class diagram.

5. Sequence Diagram & UML Class Diagram

Sequence diagram for One TimeStep


Task Server
Town
GlobalController
Highway
ALL Agents out of lock_out
ALL agents that are currently in lock_in
Timestep T-1 FINISH
ALL new free Agents
Timestep T-1 FINISH
Get new free Agents' next goal
Assign locks for Agents with new goals
Lock lock_out, lock_in
Free the lock_out where Agent left
Timestep T Start
Agents with new goal & locks
Timestep T START
Agents currently in lock_in
(not needed) Timestep T START
EECBS
Handle Agents from L2
Do the path planning
Agents newly arrived in lock_out
Agent at lock_in's status
ALL new free Agents
Timestep T FINISH
ALL Agents newly arrived in lock_out
(actual) Timestep T START
ALL Agent at lock_in's status
Get the Agents newly arrived in lock_out
Handle the Agents at lock_in
ALL Agents out of lock_out
Timestep T FINISH
ALL Agents that are currently in lock_in
Get new free Agents' next goal
Assign locks for Agents with new goals
Lock lock_out, lock_in
Free the lock_out where Agent left
Timestep T+1 Start

These are Timestep (T-1)'s postprocessing, more note can be found on Timestep T's postprocessing.

As list of pairs of (a_id, a_state) Town should then use EECBS to route these agents to their corresponding destinations (if lock_out is null then route to goal, otherwise route to lock_out)

These two can be sent in ONE lock (one message after another with nothing in between). Global send EACH Town these two messages to indicate timestep start.

Town should check if any of these agent's access_cells is free (After running EECBS on existing Agents). If so, Town takes the agent, and returns "True". Otherwise don't do anything and the road will assume the placement fails

This is not an actual message (not needed) since Highway don't need any more information from Global to start planning the current agents. And should only update agent informaiton after receiving more information about newly arrived agents in lock_out and agent at lock_in's status.

Town routes all Agents in town based on EECBS plan, ignoring agents in locks.

Town checks if any of these agent's access_cells is free. If so, Town takes the agent, and returns "True". Otherwise don't do anything and the road will assume the placement fails

Finding paths for each Agent currently at lock_out (to their lock_in). Do conflict solving and determine each Agent's next move to route all Agents currently in Highway.

For Agents newly arrived at lock_out's access cell, Town should update their information to lock_out, and send it to Global blindly, assuming the lock_out is free. Global will then pass it to the Highway.

These three can be sent in ONE lock (one message after another with nothing in between). EACH Town send these three messages to Global to indicate timestep finish (for this particular town) since nothing can be down after this before recieving more information from Global.

Town tells global the Agents that successfully entered the access cell. It means the Agents are now in Town's control, so the Highway can get rid of it.

Global tells Highway the Agents newly arrived in lock_out. Highway does not need to do anything other than blindly take the agent, because lock_out must be free.

These two can be sent in ONE lock (one message after another with nothing in between). Glabal send Highway these two messages after getting these information from ALL Towns.

Global tells Highway the Agents that successfully entered the access cell. It means the Agents are now in Town's control, so the Highway can get rid of it.

Get the Agents that newly arrived in lock_out(sent by the towns). These are the Agents that will be added to path finding in the next time step.

At the end of last time step, Highway will send the list of Agents at lock_in to the global controller. At the start of this time step, global will ask towns to check if the access cells associated with the locks are available. For each Agent, if Highway receives the message that it successfully entered the access cell, Highway will remove it from Highway system. Otherwise, keep it in the lock_in in Highway system (request to enter in the next timestep, and repeat).

These two can be sent in ONE lock (one message after another with nothing in between). Highway send Glabal these two messages to indicate timestep finish.

Besides Agents currently in lock_in, also notify Global Agents that just leaves lock_out, so Global can free the lock_out.

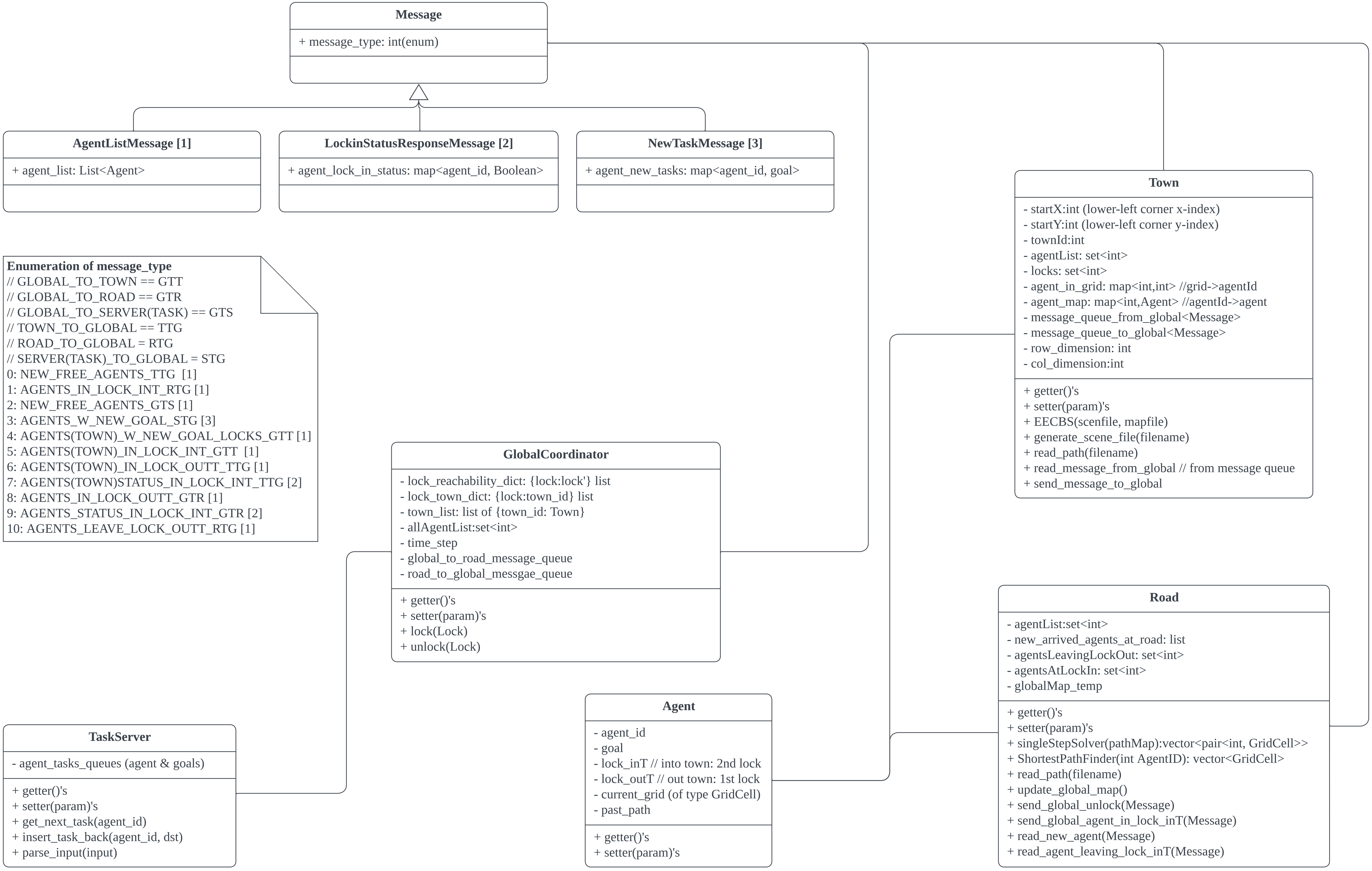
After each timestep finish for a Town, it send global a list of ALL agents that are currently free (arrived finish). This is also a message of FINISH.

After each timestep finish for Highway, it send global a list of ALL agents that are currently in lock_in (along with their access_cells). This is also a message of FINISH.

If Agent does not have next goal, should remove agent from map. For now assume Agent has next dst F

if no locks exist, store the task back to task server (indicating not assigned), assign a random F' in T1 as goal so the agent simply move around in the original town waiting until next time to check lock

Lock means now the locks are not free locks anymore. Free means the locks are now free locks again.



Message

+ message_type: int(enum)

AgentListMessage [1]

+ agent_list: List<Agent>

LockinStatusResponseMessage [2]

+ agent_lock_in_status: map<agent_id, Boolean>

NewTaskMessage [3]

+ agent_new_tasks: map<agent_id, goal>

Enumeration of message_type
// GLOBAL_TO_TOWN == GTT
// GLOBAL_TO_ROAD == GTR
// GLOBAL_TO_SERVER(TASK) == GTS
// TOWN_TO_GLOBAL == TTG
// ROAD_TO_GLOBAL = RTG
// SERVER(TASK)_TO_GLOBAL = STG
0: NEW_FREE_AGENTS_TTG [1]
1: AGENTS_IN_LOCK_INT_RTG [1]
2: NEW_FREE_AGENTS_GTS [1]
3: AGENTS_W_NEW_GOAL_STG [3]
4: AGENTS(TOWN)_W_NEW_GOAL_LOCKS_GTT [1]
5: AGENTS(TOWN)_IN_LOCK_INT_GTT [1]
6: AGENTS(TOWN)_IN_LOCK_OUTT_TTG [1]
7: AGENTS(TOWN)STATUS_IN_LOCK_INT_TTG [2]
8: AGENTS_IN_LOCK_OUTT_GTR [1]
9: AGENTS_STATUS_IN_LOCK_INT_GTR [2]
10: AGENTS_LEAVE_LOCK_OUTT_RTG [1]

Town

- startX:int (lower-left corner x-index)
- startY:int (lower-left corner y-index)
- townId:int
- agentList: set<int>
- locks: set<int>
- agent_in_grid: map<int,int> //grid->agentId
- agent_map: map<int,Agent> //agentId->agent
- message_queue_from_global<Message>
- message_queue_to_global<Message>
- row_dimension: int
- col_dimension:int

+ getter()'s
+ setter(param)'s
+ EECBS(scenfile, mapfile)
+ generate_scene_file(filename)
+ read_path(filename)
+ read_message_from_global // from message queue
+ send_message_to_global

GlobalCoordinator

- lock_reachability_dict: {lock:lock'} list
- lock_town_dict: {lock:town_id} list
- town_list: list of {town_id: Town}
- allAgentList:set<int>
- time_step
- global_to_road_message_queue
- road_to_global_messgae_queue

+ getter()'s
+ setter(param)'s
+ lock(Lock)
+ unlock(Lock)

Road

- agentList:set<int>
- new_arrived_agents_at_road: list
- agentsLeavingLockOut: set<int>
- agentsAtLockIn: set<int>
- globalMap_temp

+ getter()'s
+ setter(param)'s
+ singleStepSolver(pathMap):vector<pair<int, GridCell>>
+ ShortestPathFinder(int AgentID): vector<GridCell>
+ read_path(filename)
+ update_global_map()
+ send_global_unlock(Message)
+ send_global_agent_in_lock_inT(Message)
+ read_new_agent(Message)
+ read_agent_leaving_lock_inT(Message)

Agent

- agent_id
- goal
- lock_inT // into town: 2nd lock
- lock_outT // out town: 1st lock
- current_grid (of type GridCell)
- past_path

+ getter()'s
+ setter(param)'s

TaskServer

- agent_tasks_queues (agent & goals)

+ getter()'s
+ setter(param)'s
+ get_next_task(agent_id)
+ insert_task_back(agent_id, dst)
+ parse_input(input)

6. Current Main Components & Status

6a. Tigo's Town Component:

Inputs:

- Scenario files for each town(specifying agents' goal in each town)
- Map files and basic information(townID, dimension…) for each town
- All agents in the town

Outputs:

- The path for each agent in each town for each timestep from EECBS

6b. Ricardo’s Highway Component (Two main functions):

1. Shortest Path Finder:

Inputs:

- The map/graph of highways
- Agent newly enters the highway from towns(also with its current and goal location, which are locks)

Outputs:

- The shortest path from the agent’s current location to the goal location is a vector of (x,y) coordinates. Store the path plan for the agent.

2. Single Step Solver:

Inputs:

- All agents on highways, with their shortest paths that have been planned for them

Outputs:

- The next step of each agent. (By taking the next step from each agent’s path plan, check if there is any conflict. Solve the conflict and decide the priority based on some traffic rules. If an agent can move its next step, update its path plan by deleting this step from its path plan)

6c. Dongming’s Highway Layout Algorithm Component:

Input & Output of the Algorithm (written in Python):

Inputs:

- The MAPF baseline map (.map file)
  - The whole original ascii map
  - (.) as empty grid, (others) as obstacles
- The config file, where we specify some hyper-parameters:
  - max_town_size: the expected maximum town size
  - max_search_around: maximum L1-distance to search around for endpoints
  - visited_add_cost: how much cost added to edges when lay highway
  - max_allowed_cost: max cost allowed before blocking the edge in highway usage

Output:

- Highway File: (txt)
  - Highway nodes & Locks as ascii map
  - (s, n, w, e) as directions, (I) as intersection, (@) as obstacle/town, (L) as locks
- Town File: (json), for each town in the map, the file have a dict of its:
  - Town index
  - Town nodes & Locks as ascii map
  - (.) as normal town node, (@) as obstacle/highway, (L) as locks
  - Origin (left bottom corner’s index)

Steps of the Algorithm:

1. G: read the original graph from input MAPF map file, only keep the largest connected component subgraph as our interest (otherwise, connectivity cannot be satisfied).
2. G_d: the directed version of G, 2 opposite direction edges for each original edge.
3. ideal_endpts: from the G's dimension and max_town_size, the ideal endpoints (not necessarily nodes) of highways so that the highways will partition the map in a way that each town has size around max_town_size.
4. actual_endpts: from ideal_endpts and max_search_around, search around the ideal endpoints for actual endpoints that should be actual nodes in the map to be highway endpoints.
5. outer_endpts_list: the outermost endpoints around the map of actual_endpts, in clockwise direction order.
6. outer_highways: plan highways between the outer_endpts_list in clockwise direction order (this aims to increase connectivity).
7. inner_highways: plan highways between each adjacent actual highway endpoints in actual_endpts.
8. Post Processing the highways and create some graphs: G_hw: graph only storing highway nodes; G_together: graph storing highway and town nodes; G_temp_locks: temporary lock locations (might be invalid); G_final_town: contains all non-highway nodes.
9. G_valid_locks: from the G_temp_locks save only the valid locks (each valid lock should be reachable from town nodes and should not disconnect the town).
10. Finally, post processing the result and create the output files.

Versions and Accomplishments in each Version:

v1, v2: not working

v3: graph edge is unweighted

- highly possible to find path with overlapping edges

v4: graph with weighted edge design

- problem: edge cases where limiting edges does not resolve conflicts at intersection
- simply removing edge/adding edge cost does not resolve node conflicts

v5: weighted & directed edge design

- when creating the original graph, each undirected edge replaced by 2 directed opposite edge (conjugate)
- when an edge is occupied by a hw, remove its conjugate => avoid actual conflict
- when an edge is occupied by a hw, add cost c in [0, inf) => avoid too much traffic
- in this way, each road can have at most 1 in/out in each direction, a max of 4 in/out

v6: add file output function

v7: add "lock" function: set lock as L in the output files

v8: what "lock"s each "lock" can reach; replace end of road with X

v9: try to fix previous lock's two problems:

1. lock not connected to any town nodes => cannot reach from town
2. lock disconnect town nodes => town become disconnected now

v10: refactor code & add documentation

v11: modifications on making the result T-connected

1. find outer highway endpoints

2. try to make them connected clockwise, before layout other part of highways
3. * this step can be done recursively, layer by layer, possible extension...

- **Definition of T-connected:** for each town1-town2 pair, there is a direct highway from town1 to town2, where:
    - adj 2 town nodes (within same town) are bi-direction connected
    - adj town node & lock node are bi-direction connected
    - adj 2 lock nodes are NOT connected
    - adj lock node & highway node are bi-direction connected
    - adj 2 highway nodes are only connected based on highway direction

v12: build town classes, check town connectivity

v13: add more constraint on picking outer highway endpoints

1. the end point picked is as close to the ideal as possible (max is max_search_around)
2. the end point picked MUST be connected with 4 surrounding map nodes

- result T-connected: Boston_0_256 (0 unconnected), ost003d (0 unconnected), den312d (0 unconnected)， random-64-64-20 (0 unconnected), den520d (1 unconnected)

v14: improve inner highway layout algorithm:

1. try all adjacent (s, t), same as before
2. if not found path between (s, t), try: (s-, t) & (s, t+) until find both or run out of s- & t+