Ferrari Master Plan Project Report

Davide Cavicchini, Laurence Bonat

May 2024

1 Introduction

In this report, we explore the key components of our agent, which is designed using the Belief-Desire-Intention (BDI) architecture. Our goal is to explain how the agent adapts to changing environments by continuously updating its knowledge, adjusting its goals, and planning its actions to achieve these goals effectively. This flexibility is essential for the agent to navigate complex situations where the environment and its objectives can change rapidly.

The following sections will break down the various parts and processes that make our agent work. We will look at how the agent updates its knowledge, chooses and carries out its goals, and the planning methods it uses to decide on the best actions. Additionally, we will discuss the specific parameters that control these processes, which allow us to fine-tune the agent's behavior for different situations.

In this report, we will focus exclusively on the logic used by the collaborative version of our agent. However, since the agent is designed to ignore all collaborative optimizations until it handshakes with another agent, the code also functions in single-agent mode.

2 Beliefs

To react rapidly to environmental changes, our agent updates its beliefs asynchronously using the listeners provided by the Deliveroo game. This setup allows continuous sensing and processing of information about surrounding agents and parcels, maintaining an accurate and dynamic game state. In addition, we keep a live representation of the entire map packed with all our knowledge of the current and predicted game states, which we leverage in our planners and control loop.

2.1 Belief Revision Loop

The Belief revision is separated into three macro elements: agents, parcels, and the map. Each has to edit the beliefs for that particular element of the game, while giving access to all the other components to the created representations that should be kept consistent, avoiding race effects.

2.1.1 Parcel sensing

The "onParcelsSensing" listener detects parcels each time they enter the sensing range. For each we store the score, carried state, and position; then the score is decreased every $PARCEL\_DECAY\_INTERVAL$ ms, computed from the game configuration. We also check if a previously detected parcel should be in view to know if another agent picked it up, thus having to remove it from the stored parcels. We apply the same approach to parcels received by the collaborative agent that fall outside our sensing range. This data is stored in a map of parcels that uses their IDs as keys to identify them.

Additionally, we store in a separate registry the agent who carries the parcels, which is useful for agent predictions since we can try to infer if an agent intends to deliver a parcel or not, as we will see in the next section.

2.1.2 Agent sensing

Upon receiving the "onAgentsSensing" event, we iterate through the received agents and the agents sent by our collaborator. For each, we save their history up to $MAX\_HISTORY$ times and try to predict their moves up to MAX_FUTURE move. The prediction only examines the last move: if the move is not directed towards a parcel or the agent is not carrying one, it continues in a straight line. Otherwise, we use the planner to generate a path to the parcel or the closest delivery zone. We also have a logic to invalidate an agent when I predict a wrong move, and a logic to remove it from the map if I do not see it for $DELETE\_UNSEEN\_AGENTS\_INTERVAL$ ms, so that it does not block valuable tiles since it is highly unlikely it stayed in the same position and it might be blocking optimal paths. The collaborative agent is saved as a normal agent, but it also has additional information, like its actual plan, which we will talk about in detail in section 3.

Each agent is modeled using a class containing its position, history, believed intention, and future moves. This class is made accessible to the rest of the agent program through a map of agents that uses their IDs as keys.

For PDDL, we also create the belief set of the current position of the agents and the predicted ones, these will be used in the planners as we will see in section 5.2.

2.1.3 Map

The map is fundamental because we access it for our intention revision and planning, so it has to be in the least inconsistent state each time. For this reason, the map updates its state at a fixed time interval set as the movement speed. In this way, the agent should always have an updated game state. Each update is done on a separate copy and overwrites the old map data only at the end of the update process.

When the agent connects, we receive all the information on the structure of the map and use it to instantiate the map tiles and compute all the needed heuristics, such as the spawn-able regions, the distance to the delivery for each tile, and others. The structural information is also saved in a belief set to be used in the PDDL planners.

At each iteration, the map is updated with all the current agents and parcel positions, so we get an accurate representation. At the same time, we keep an array of predicted maps, with the predicted movements of the agents in the future, which will be useful for some search algorithms we look at in section 5.1. In the map, we also update the timestamp in which we last saw a specific tile, saved as seconds passed from the game start, used in the explore planner as we will look at in section 5.6. But to make it behave correctly, we needed to put a higher limit to it, MAX_TIME, after which we rescale the timestamps by $LAST\_SEEN\_RESCALE\_FACTOR$. This is done to avoid throwing away all the information collected to date, while still managing their values. Lastly, we generate a heatmap for the agents, which is also used in the explore planner, where each agent influences an area with a radius of $MAX\_AGENT\_HEATMAP\_DISTANCE$, adding one to the heat of the tiles within this area, disregarding obstacles.

2.2 Parameters

Looking at the Belief revision we have control over:

- **MAX_FUTURE**: Increasing it is useful for maps with long and narrow corridors since the planner would be able to take a path based on the assumption of the agent preceding us clearing the passage.
  Unfortunately, using PDDL, this parameter has to be kept low, otherwise most of the time we would not find a plan.
- **MAX_SPAWNABLE_TILES_DISTANCE**: This parameter controls the region segmentation used to grow the spawn-able zones. This helps in the case of sparse spawn-able tiles that should be considered as a single zone.

- **MAX_AGENT_HEATMAP_DISTANCE**: For maps with narrow corridors, a small value is preferable since the current implementation does not check if the affected tiles are immediately reachable, while for wide open maps, the agent can take many paths to a zone so it is useful to increase the radius to take them into account.
- **MAX_HISTORY**: used to increase the memory of the last agents' moves. It does not affect our agent behavior since the predictions only look at the last move of each agent.
- **DELETE_UNSEEN_AGENTS_INTERVAL**
- **MAX_TIME**
- **LAST_SEEN_RESCALE_FACTOR**

3 Communication

In this section, we will see the main characteristics of the communication between two coordinating agents.

Communication begins by identifying the collaborator agent's ID through a broadcast handshake message (see Figure 2). The handshake is accepted if the other agent's name contains the NAME parameter string. Post-handshake, agents can exchange messages directly. This procedure is also used to assign the roles: Agent 0, the master, is the one who receives first the handshake message and responds with an "ACK" message, the other is agent 1.

The agents talk to each other through the game messages and requests, which are accumulated into the according buffer, to be read when the agent is ready. We have four different circular buffers:

- the $agentBuffer$ for the shared agents that our collaborator sees
- the $parcelBuffer$ for the shared parcels that our collaborator sees
- the $requestBuffer$ for incoming requests from the collaborator
- the $awaitBuffer$ for separating *await* requests from the other types of requests

3.1 Belief Sharing

The beliefs are shared by sending a message with a header indicating the type of information arriving, so if it is information about the collaborator or a belief, and a content field containing another header specifying the sub-category of the information we are sharing, being agents or parcels for the beliefs, and plan, intention, or position for the collaborator information. The general protocol is illustrated in figure 3.

The received beliefs are stored in the corresponding buffers to be read by the belief revision, while the collaborator information overwrites the old one.

3.2 Requests

Requests are identified by the main header being "request", and uses a simple protocol to handle them by checking the content of the message to separate *await* requests into the $awaitBuffer$ for the synchronization and the others into the $requestBuffer$ for the coordination.

When sending a request we wait for an answer for a maximum of $MAX\_REQUEST\_TIME$ ms, if I receive nothing back I consider the communication failed.
When waiting for a request, we check the $requestBuffer$. If there is more than one request in the buffer we only accept the last one, assuming it is the only valid request. While, if there are no requests in the buffer, we retry after 100ms for MAX_AWAIT_RETRY times maximum.

Synchronization A special type of request is the await and answer, used by specific moves that help us synchronize the plans. The *await* move sends an *await* request and waits for an answer. The *answer* move answers to the await requests present in the buffer. As for all requests, if we do not receive a response from the collaborator it will time out. Prompting us to proceed with our plan, assuming they have encountered some issues. In Figure 4, we can observe the general protocol between two agents.

3.3 Parameters

To coordinate our agents we can control:

- **NAME**
- **MAX_MSG**: defines the size of the buffers
- **MAX_REQUEST_TIME**
- **MAX_AWAIT_RETRY**

4 Agent Control Loop

In alignment with the BDI architecture, our agent follows the following control loop: decide about possible intentions to adopt and revise the Intention set I, select the new intentions to adopt from I, generate a plan P for it, and execute it. To be as reactive as possible to changes in the environment the first two steps are executed asynchronously from the intention execution.

4.1 Intention revision

We decided to have three possible types of intentions:

- *Explore*: This intention is used as a "fallback" when we do not know what to do, so we explore the map.
- *Pickup*: Intention to take a parcel. We can have multiple pick-up intentions simultaneously, each for a different parcel targeted by the agent.
- *Deliver*: Intention to deliver the parcels we are currently holding.

When the agent starts, the only two types of intentions it will have are to explore and to deliver. After each $INTENTION\_REVISION\_INTERVAL$ ms, the agent will revise its intentions, and if the belief revision has detected new parcels, an intention to pick up each of them will be created.

4.2 Intention selection

To choose which intention our agent should pursue we define functions that assign a utility to each.

Since we wanted the *Explore* intention to be a fallback when there is nothing better to do we return a constant value of 0.1.

Regarding the *Deliver* intention, we use the expected reward, defined simply by summing up the current parcels' scores and subtracting the points we lose to get to the closest accessible delivery point. So with C being the set of carried parcels:

$$DeliverUtility = \sum_{p \in C} \max(0,\ p.score - loss\_to\_deliver)$$

For the *Pickup* we want to give precedence to the parcels that yield the highest reward when delivered, while also considering the other agents' positions, and giving some weight to the risk we are taking when picking up a parcel while carrying others. To do so we compute for each parcel the

plan to get there, taking into account also parcels along the way, sum up the scores of the parcels, then calculate the plan to get to the closest accessible delivery, and use the total steps to calculate the points we lose. To account for the risk, we multiply the number of steps needed by a *penality*, controlled by a parameter as we will see in the dedicated section, to compute the loss of points for the carried parcels:

$$SPickupUtility = \sum_{p \in C} \max(0,\ p.score - (loss\_Pick + loss\_Del) * penality) + \sum_{p \in Path} \max(0,\ p.score - (loss\_Pick + loss\_Del))$$

For parcels on the same tile of an agent, we return a utility of -1, and 0 for parcels closer to our collaborator. Regarding the parcels where other agents are closer than us, we wanted them to be chosen over the *Explore* intention while also being discarded in favor of safe parcels. So, we decided to make use of the interval $[0.2, 1]$: the base score of an unsafe parcel will be 0.2, then based on how much closer the other agent is we get a number in the interval $[0, 0.3]$, finally we use the score of the parcel to calculate a number between $[0, 0.5]$, summing all of them up we get the utility of the intention to pick up that parcel.

$$UPickupUtility = 0.2 + \frac{p.score}{2(avg\_score + var)} + 0.3\frac{(my\_steps - adv\_steps)}{map.w + map.h}$$

Since we want our utilities to be as precise as possible, we incorporated two more tunings:

1. To compute the points lost from having to move N steps, we not only take into account the movement duration provided by the game configuration but also compute an exponentially decaying mean of the actual time it took us to move, we call the difference between the two *move_slack*. The computation of this mean is detailed in the Parameters section, while for the loss:
$$loss\_steps = \frac{N * (MOVEMENT\_DURATION + move\_slack)}{PARCEL\_DECAY\_INTERVAL}$$
2. If the intention is not currently being executed, we also subtract from the parcel scores the time taken by the planner, which we call *planning_time*. This value is also approximated using an exponentially decaying mean as we will see in the Parameters section, and it contributes to the loss as follows:
$$total\_loss = loss\_steps + \frac{planning\_time}{PARCEL\_DECAY\_INTERVAL}$$

4.3 Intention execution

When an intention is selected, we send a signal to stop the current one and wait for it to begin the new one, for which a new plan is created using the appropriate planner and then executed.
The execution of the plan goes as shown in Algorithm 1.

4.4 Parameters

For this section of our agent, we can control the following parameters:

- **MAX_RETRIES**: Regulates how "stubborn" our agent is, this can be helpful in narrow corridors hoping for the other agent to yield, and to avoid deadlocks of "left-right" between two opposing agents.
- **HARD_REPLAN_MOVE_INTERVAL**: Controls the reactiveness of our agent to changes in the environment, but it has to be carefully calibrated with the previous parameter.

```
for i ← 0, P.length do
    res ← P[i].execute();
    if ¬res then
        i − −; retryCount + +;
        if retryCount ≥ MAX_RETRIES then
            if stop then
                stop execution
            end
            await [1, MAX_WAIT_FAIL] moves;
            i ← 0; P ← recoverPlan();
            if ¬P then
                P ← replan(P);
            end
        end
    else
        retryCount ← 0;
        if i mod HARD_REPLAN_MOVE_INTERVAL = 0 then
            if stop then
                stop execution
            end
            i ← 0; P ← replan();
        end
        if i mod SOFT_REPLAN_INTERVAL = 0 then
            i ← 0; P ← beamSearch(P);
        end
        if i mod CHANGE_INTENTION_INTERVAL = 0 & stop then
            stop execution
        end
    end
end
```

Algorithm 1: Intention execution loop

- **SOFT_REPLAN_INTERVAL**: This enables us to avoid calling the PDDL planner for deviations that are trivial to compute such as picking up parcels on the way.
- **CHANGE_INTENTION_INTERVAL**: This also helps the agent be more reactive, but it works better with PDDL since it does not always require to re-call the planner, only when needed.
- **INTENTION_REVISION_INTERVAL**
- **STOP_WHILE_PLANNING_INTERVAL**: Since PDDL planning might take a long time, this allows the agent to be more reactive, by sending another planning request without waiting for the previous one to end.
- **PENALITY_RATE_CARRIED_PARCELS**: Quantifies the "risk" of trying to pick up another parcel while already carrying something. It is the value assigned to the *penality* multiplier we talked about.
- **BASE_PLANNING_TIME** and **PLANNING_TIME_DECAY**: Controls the exponentially decaying mean of the time taken by the planners, initialized using the provided parameter and updated using: $planning\_time = planning\_time * decay + (1 - decay) * time\_taken\_by\_planner$
- **BASE_MOVE_SLACK** and **SLACK_DECAY**: similarly, we do the same for the surplus time the server takes to execute a move.

5 Planning

In this section, we will explore how our agent plans in the Deliveroo environment. First, we will introduce the base algorithms used by the agent and their counterpart in PDDL. Then, we will explain how they are combined to create the plans to reach the selected intention.

5.1 Search algorithms

To search for an optimal path to a goal, we use four variations of Breadth-First Search (BFS): Clean BFS, Frozen BFS, and Future BFS. Given the small map sizes and absence of memory constraints, BFS finds the optimal path without the complexity of other algorithms. In addition, we use a Beam Search to include the pickup actions into the path.

5.1.1 Clean BFS

This version of the BFS algorithm considers only the physical boundaries of the map and does not consider any obstacles like other agents on the map. It provides a clear path between any two points on the map if one exists.

5.1.2 Frozen BFS

This BFS captures the existing map's state, seeking a path that cleverly utilizes the current locations of the other agents. Therefore, if an agent is located on the optimal route between the starting point and the objective, it avoids exploring the occupied tile by finding an alternative path, if one exits.

5.1.3 Future BFS

The future BFS exploits the information in the predicted maps, built as described in Section 2.1. We start at time $t = 0$. When selecting a tile to explore, we only consider tiles available at time $t+1$ and not conflicting with the collaborator's plan. When we reach the final predicted timestamp, we switch to a frozen BFS using the last predicted map. This version also allows the agent to use a "waiting move", where it will wait for the other agents to move while staying in the same tile, hoping for a better path to clear.

Using this method, we can use predictions to follow another agent through a narrow corridor, as we predict it will clear the passage before us. We can also use these predictions to avoid paths that may become blocked in the future; for instance, this is helpful when we need to pass through a bottleneck, as we can predict if and when it will be blocked.

5.1.4 Beam Search

This algorithm is designed to deviate from a set path and explore potential parcels to pick up within a "corridor". Given a path, the explorable tiles are determined as a corridor around it deviating from the original by 1 tile in each direction. If a parcel is found within these tiles, a deviation is added to pick it up.

5.2 PDDL planners

For PDDL we re-implemented the same logic as in their algorithmic counterparts.

In addition, we decided to try and implement a planner to directly predict a path to pick up a parcel and deliver it in a single planner call using the frozen logic. Since the latency was consistent regardless of the complexity of the plan we hoped to cut in half the planning time. Unfortunately, in most maps, this planner failed to find a plan that reached both objectives; forcing the agent to use the other planners, resulting in even more calls. We still decided to include it in this report, since it was a stimulating problem to solve in PDDL.

Map representation All the planners we devised share the same representation of the map and the position of our agent. The needed information is encoded using three propositions: **at** ?tile, **visited** ?tile, and **connected** ?from ?to. The latter is maintained inside the map introduced in section 2.1.3 and is a constant throughout the resolution of the problem, while the other two are instantiated and updated as needed.

5.2.1 Clean Search

Problem The problem only contains the basic prepositions for the map representation and our agent position, initialized as needed.

Domain It only contains one action to move from one tile to another, taken as parameters, checking if the two are connected and if the destination is not already visited. The effect is to move the agent, modify the "at" preposition, and set the destination tile to visited.

Objective The objective is simply a disjunction of (at goal_tile_i).

5.2.2 Frozen Search

Problem It is similar to the previous one, in which we add the "agent ?tile" preposition to encode the tiles occupied by other agents.

Domain It also has only one action to move, but it also checks for the destination tile to be free of agents.

Objective The objective is the same.

5.2.3 Future Search

Problem For this, we change the representation of the occupied positions by other agents, by adding to the preposition the timestamp where that information is valid "agent ?tile ?time". We also consider the plan of the collaborator using the preposition "collaborator ?tile". Finally, we encode the current timestamp with the preposition "time ?timestamp". To implement the logic of waiting at the tile for a maximum of N times, we also need N prepositions "waited$i$?tile". For an example look at the appendix.

Domain It still has the move action, but we need one for each timestamp! The action to move at time t makes the same checks as the clean version, but then also checks for "not (agent ?to $t+1$)", for "not (collaborator ?to)", and for "time t". The effect moves the agent, sets the visited, and increases the timestamp "time $t+1$". Finally, we have the waiting move, one for each waited repeated for each timestamp. Looking at an example, the action "wait1T1" will check for "time 1", "not (waited1 ?tile)", and "at ?tile"; as an effect, it will increase the timestamp "time 2", and set "waited1 ?tile" to true. For an example look at the appendix.

Objective The same as the previous ones.

5.2.4 Pickup + Deliver

Problem This problem is similar to the frozen search, but we have two different prepositions to encode the visited information for the two searches to and from the parcel. We also introduce two 0-ary prepositions to signal when the two goals, pickup and deliver, are reached.

Domain We have two move actions, one for the trip to the parcel and one for going back to the delivery zone, they behave similarly but check for different visited, and the latter needs the "pickup" to be true to be executed, ensuring the planner does not waste moves until we have to make our way back. Then we have an action to pick up the parcel where we check to be on the parcel tile, and for "not (pickup)", so we can make it true. Finally, a move to deliver checking to be on one of the delivery tiles and for "pickup" to be true, if so it makes "deliver" true.

Objective The objective is reached when we have picked up and delivered the parcel, so we only have to check for the homonym predicates to be true.

5.3 Plan Logic

In this section, we will dive deeper into how we use these building blocks to construct the plans our agent follows to reach its intention. We decided to have three similar planners for each type of intention, being "pickup", "deliver" and "explore". In addition, when implementing PDDL we added two components: a soft planner which calls the beam search on the current plan path to pick up parcels that might come into the sensing range, and a logic to recover a failed plan. With these we hoped to make our agent more resilient to failures, avoiding having to re-call the planner and make it more reactive to simple deviations to the plan that could increase our score.

5.4 Pickup

It is important for this planner to consistently return a path, regardless of the positions of other agents. To achieve this, a "fallback" logic is used on the search algorithm, where we first employ a future search, if it fails we use a frozen search, and as a last resort a clean search - the latter of which is guaranteed to find a solution given a reachable goal in the map. Then the obtained path is passed to the Beam Search to insert the pick-up moves into it.

5.5 Delivery

The delivery of a parcel works similarly to the Pickup planner, calling the same planners and the beam search to get to the closest accessible delivery zone. It only differs from it by appending a drop move at the end of the plan.

5.6 Explore

The explore intention does not provide a specific tile as a goal to pursue. For this reason, before exploring, the agent has to decide on an objective. To be as flexible as possible we opted to use a similar approach to the intention selection and devise a loss function that should steer our agent to tiles that we did not see for a long time, where there is a higher concentration of spawnable tiles, are relatively free of traffic, and avoid going into the same direction as the collaborative agent. To achieve this we assign the loss of a spawnable tile using:

- Last seen time: the time in which we last sensed the tile
- Tile probability: quantifying how probable it is for a parcel to spawn based on the size of the associated region.
- Tile heat: the concentration of the agents, calculated as explained in section 2.1.3
- Distance from the collaborative agent objective, ignored if we are alone

The final loss carefully weighs these contributions. It is computed as follows, rounding the result to the nearest integer:

$$\begin{aligned} Loss =& tile\_last\_seen * (1 - tile.probability) * (tile\_agent\_heat) \\ & * \left(1 - \left(\frac{distance(tile, otherAgent.intention.goal)}{map.width + map.height}\right) * \left(1 - \frac{tile\_last\_seen}{700}\right)\right) \end{aligned}$$

The path to reach the tile with the lowest loss is computed as we did for the other intentions.

However, we encounter a problem with this base formulation: when we have to revert to a clean search but other agents block all paths to the target tile, the plan inevitably fails to complete and recover. This forces us to call the explore planner again. Since we did not reach the objective tile to update its last_seen status, the planner would still choose the unreachable tile.

To avoid this, we add a $TIME\_PENALTY$ to the optimal tile each time it is chosen. This parameter allows us to penalize the tiles we cannot reach, ensuring that eventually, another tile will have a lower loss.

5.7 Plan Recover

As we saw in Algorithm 1, this procedure is called when a move fails to execute for $MAX\_RETRIES$ consecutive times. The already in place logic, which attempts the failed move multiple times, lets us try to "push" and make the other agent yield. If this does not work, we try to recover the plan without having to call the planner.

Adversarial agent If the agent who is blocking our path is not our collaborator, then we try to wait for $BASE\_FAIL\_WAIT + random(0, MAX\_WAIT\_FAIL) * MOVEMENT\_DURATION$ ms, hoping to force the other agent to back down and let us resume the old plan. If it is still blocking our path, we try to go around it by searching for a path to the location where we would be after two moves. A special case occurs if that move involves delivering a parcel; in this case, we search for a delivery tile near the target location. If even these actions fail, we commit to a hard replan and devise a new plan to get to our goal.

Coordination If instead, it is our collaborator who is blocking us, then we try to communicate with him to coordinate our efforts. To do this the master, agent 0, sends requests to be fulfilled by the slave, agent 1, which tries to fulfill them and returns the result. The general protocol is shown in image 1. We sequentially try different strategies, if everything fails or one request times out, the agent goes for a hard replan.

5.7.1 Coordination Strategies

Swap Parcels When one of the two agents is trying to deliver and is blocked by the collaborator, it probably means he is closer to a delivery tile. So, the safest bet is to swap parcels between the two coordinating agents. The pattern to match is simple: depending on the available space and who is closer to a delivery tile, one agent drops the parcels and the other picks up and delivers them. These actions are synchronized using the *await* and *answer* moves we talked about in section 3.2.

Move Aside With this strategy, one agent tries to stay still and wait until the collaborator moves out of the way. This allows the first agent to continue with its original plan, while the collaborator will return to its original tile and resume its plan. As for the swap, these actions are synchronized using the *await*/*answer* moves.

5.8 Parameters

For the planning, we chose to parameterize:

- **MAX_WAIT**: Controls for how long the future planners should consider to sit still.
- **PROBABILITY_KEEP_BEST_TILE**: Allows the collaborative agents to stochastically choose different tiles to explore when they have the same loss, particularly useful at the beginning of the game.
- **TIME_PENALTY**
- **BASE_FAIL_WAIT** and **MAX_WAIT_FAIL**

6 APPENDIX

Future Search problem file example

```
;; problem file: problem-bfs-example-problem-1.pddl
(define (problem default)
    (:domain default)
    (:objects
        t_0_0 t_0_1 [...] t_16_18 t_16_19 T1 T2 T3 T0
    )
    (:init
        (connected t_0_0 t_0_1)
        (connected t_0_1 t_0_2)
        [...]
        (connected t_16_18 t_16_17)
        (connected t_16_19 t_16_18)
        (at t_4_1)
        (visited t_4_1)
        (time T1)
        (not (time T2))
        (not (time T3)))
    (:goal
        (or (at t_10_19))
    )
)
```

Future Search domain file example

```
;; domain file: domain-CleanBFS-1.pddl
(define (domain default)
    (:requirements :strips)
    (:predicates
        (collaborator ?to)
        (time ?param)
        (at ?from)
        (connected ?from ?to)
        (visited ?to)
        (agent ?to ?param)
        (waited0 ?tile)
        (waited1 ?tile)
    )
(:action move1
    :parameters (?from ?to)
    :precondition ( and (not (collaborator ?to)) (time T1) (at ?from) (connected ?from ?to)
                    (not (visited ?to)) (not (agent ?to T1)) (not (agent ?to T0))
                    )
    :effect (and (not (at ?from)) (at ?to) (visited ?to) (not (time T1)) (time T2))
)
(:action move2
    :parameters (?from ?to)
    :precondition (and (not (collaborator ?to)) (time T2) (at ?from) (connected ?from ?to)
                    (not (visited ?to)) (not (agent ?to T2))
                    )
    :effect (and (not (at ?from)) (at ?to) (visited ?to) (not (time T2)) (time T3))
)
(:action move3
```

```
    :parameters (?from ?to)
    :precondition (and (not (collaborator ?to)) (time T3) (at ?from) (connected ?from ?to)
                    (not (visited ?to)) (not (agent ?to T3))
                    )
    :effect (and (not (at ?from)) (at ?to) (visited ?to))
)
(:action wait0T1
    :parameters (?tile)
    :precondition (and (time T1) (at ?tile) (not (waited0 ?tile)))
    :effect (and (waited0 ?tile) (not (time T1)) (time T2))
)
(:action wait0T2
    :parameters (?tile)
    :precondition (and (time T2) (at ?tile) (not (waited0 ?tile)))
    :effect (and (waited0 ?tile) (not (time T2)) (time T3))
)
(:action wait0T3
    :parameters (?tile)
    :precondition (and (time T3) (at ?tile) (not (waited0 ?tile)))
    :effect (and (waited0 ?tile))
)
(:action wait1T1
    :parameters (?tile)
    :precondition (and (time T1) (at ?tile) (not (waited1 ?tile)))
    :effect (and (waited1 ?tile) (not (time T1)) (time T2))
)
(:action wait1T2
    :parameters (?tile)
    :precondition (and (time T2) (at ?tile) (not (waited1 ?tile)))
    :effect (and (waited1 ?tile) (not (time T2)) (time T3))
)
(:action wait1T3
    :parameters (?tile)
    :precondition (and (time T3) (at ?tile) (not (waited1 ?tile)))
    :effect (and (waited1 ?tile))
)
)
```

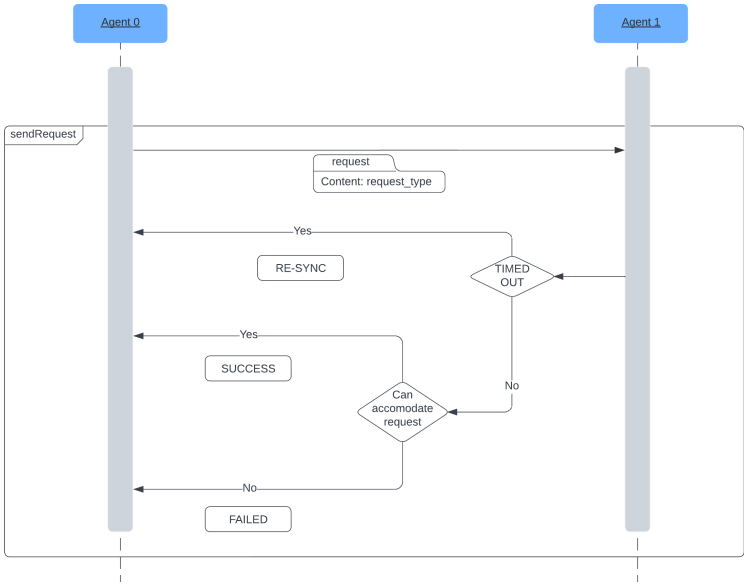

Figure 1: Coordination protocol

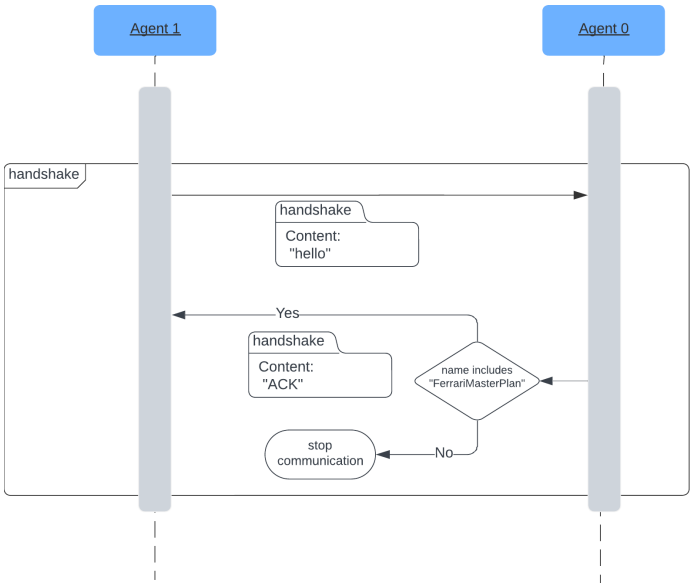

Figure 2: Handshake protocol

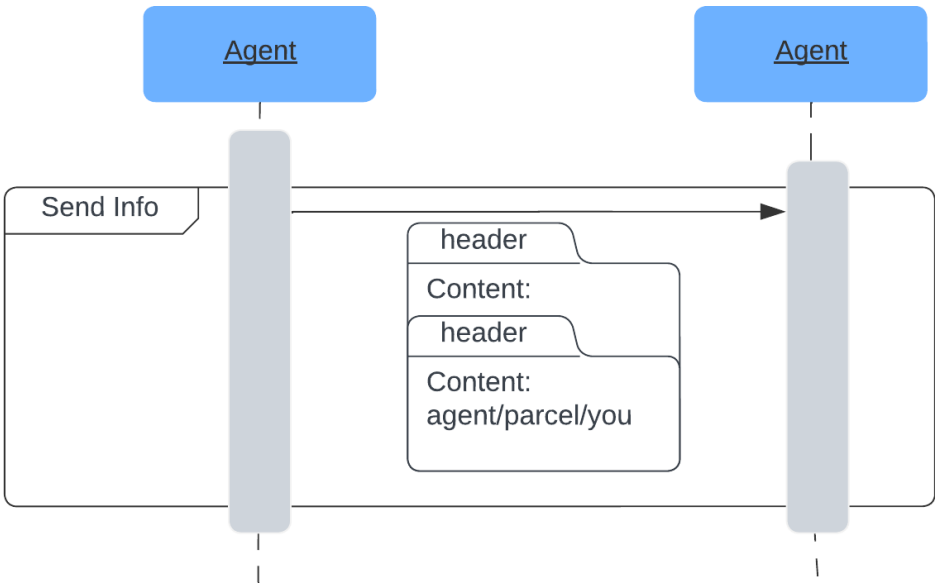

Figure 3: Protocol to send info/beliefs agent to agent

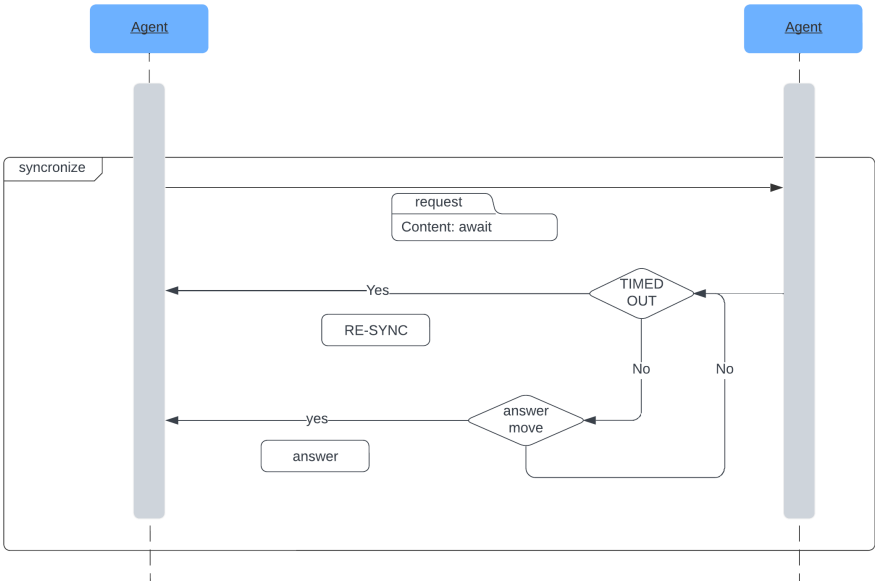

Figure 4: Synchronization protocol