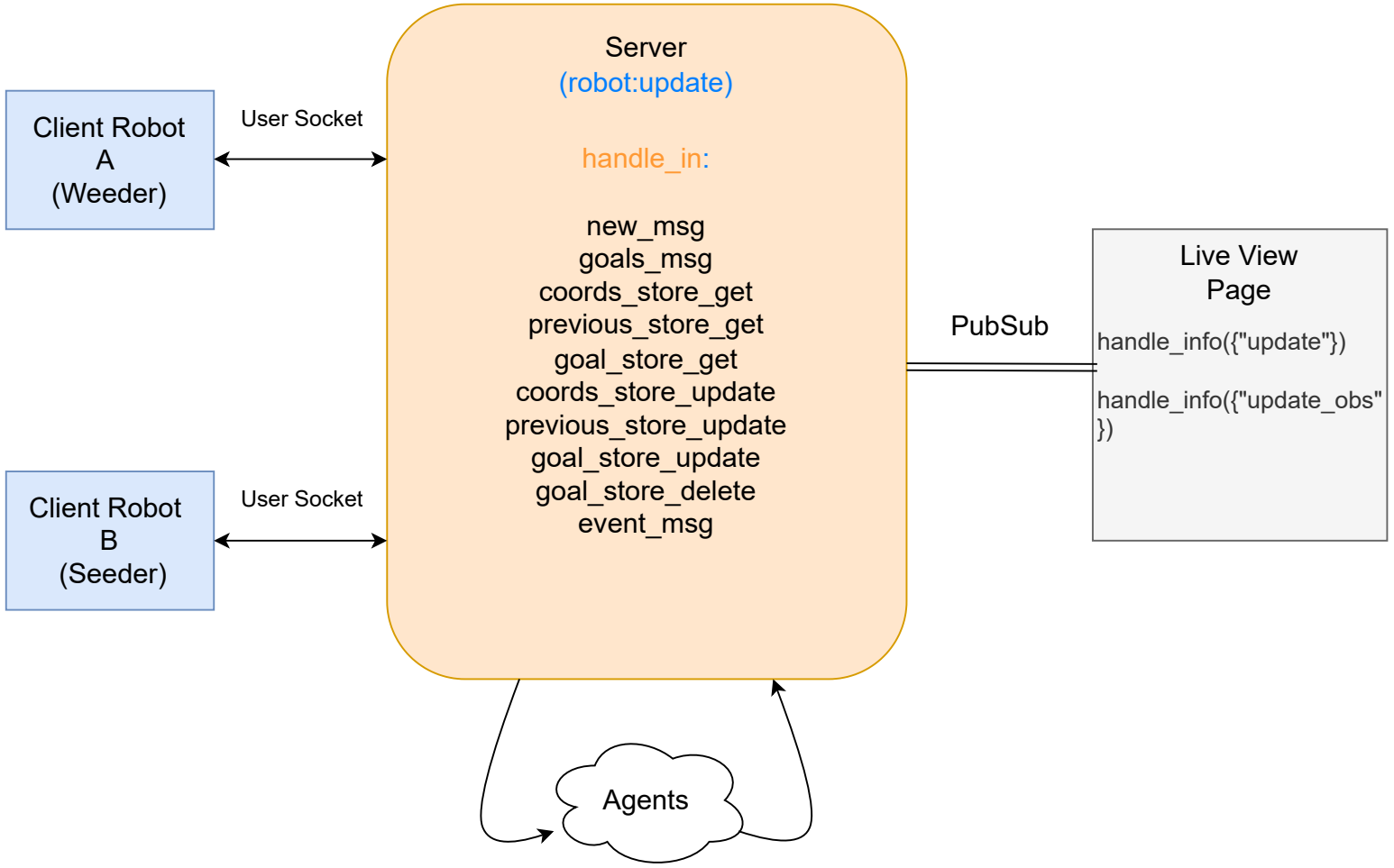
For Task 5 our team has made separate robots for seeding and weeding, with each of them navigating on a different arena. We've also used custom start positions as they were allowed.

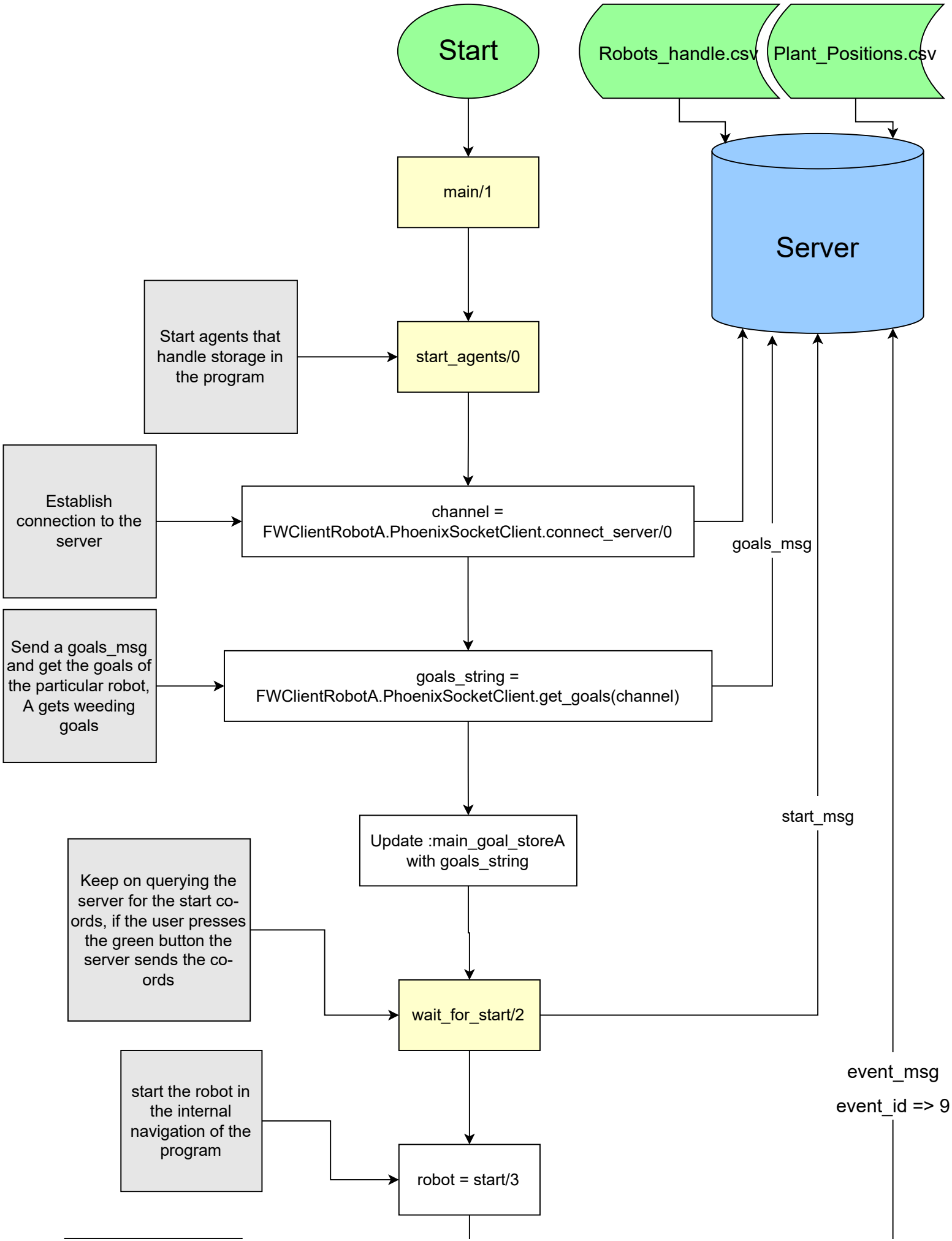
A few points to note about our submission:

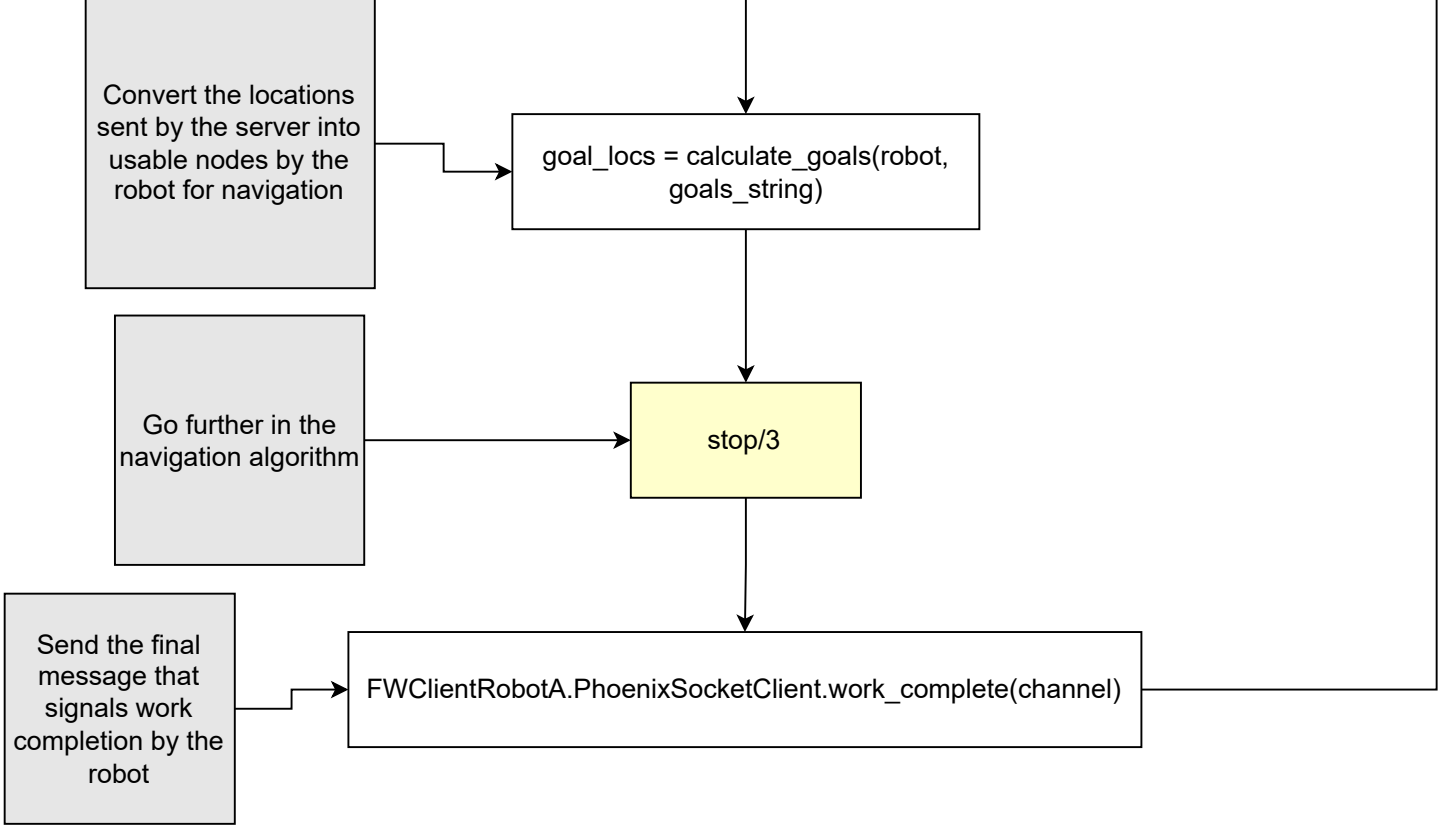
1. Our *gripper* broke down irreparably, hence we couldn't record a video of the plants being weeded, however we have shown the arm-movement and line following in detail.
2. Both the robots are able to move simultaneously however, we have uploaded videos of separate movement as our college was shut and we couldn't get a place to put down both arenas together. Other than that our seeding video shows 3/4 of the seeds being sown.
3. Coming to the algorithm, the flowchart of the whole program can be found in the attached `"Task5 Diagrams.pdf"`
4. The code for FWClientRobotB is the same as A, the only difference is instead of weeding, in the seeding function, we give angles to a servo in increments of `60 degrees`. I've mentioned it here as it isn't in the flowchart
5. We weren't able to get a way to receive the `"event_msg"` with `"event_id" => 6` sent from the Server in the Client, however the message is still being sent. We've implemented stopping the robot with a different method.

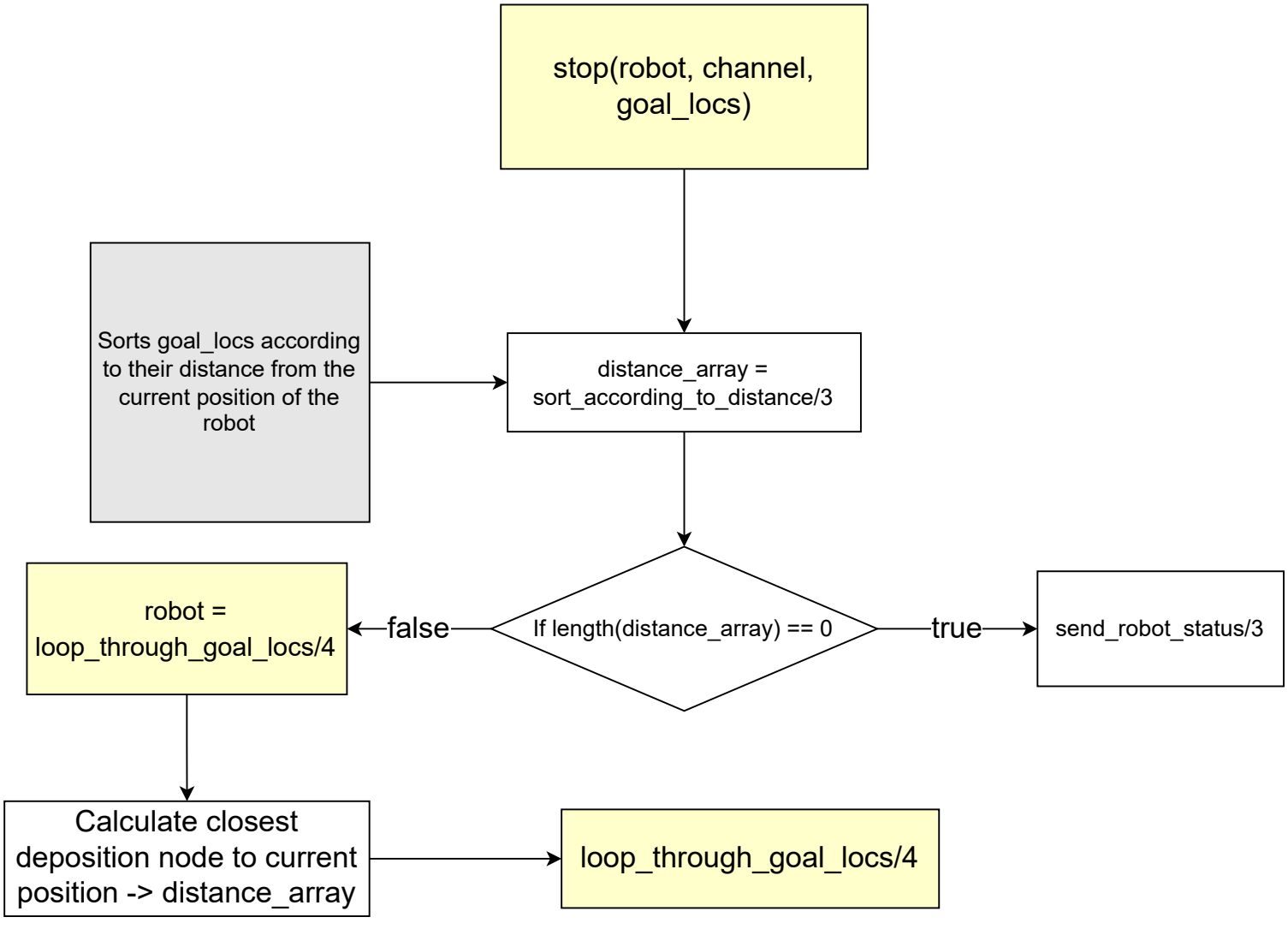
This browser does not support PDFs. Please download the PDF to view it: Download PDF.

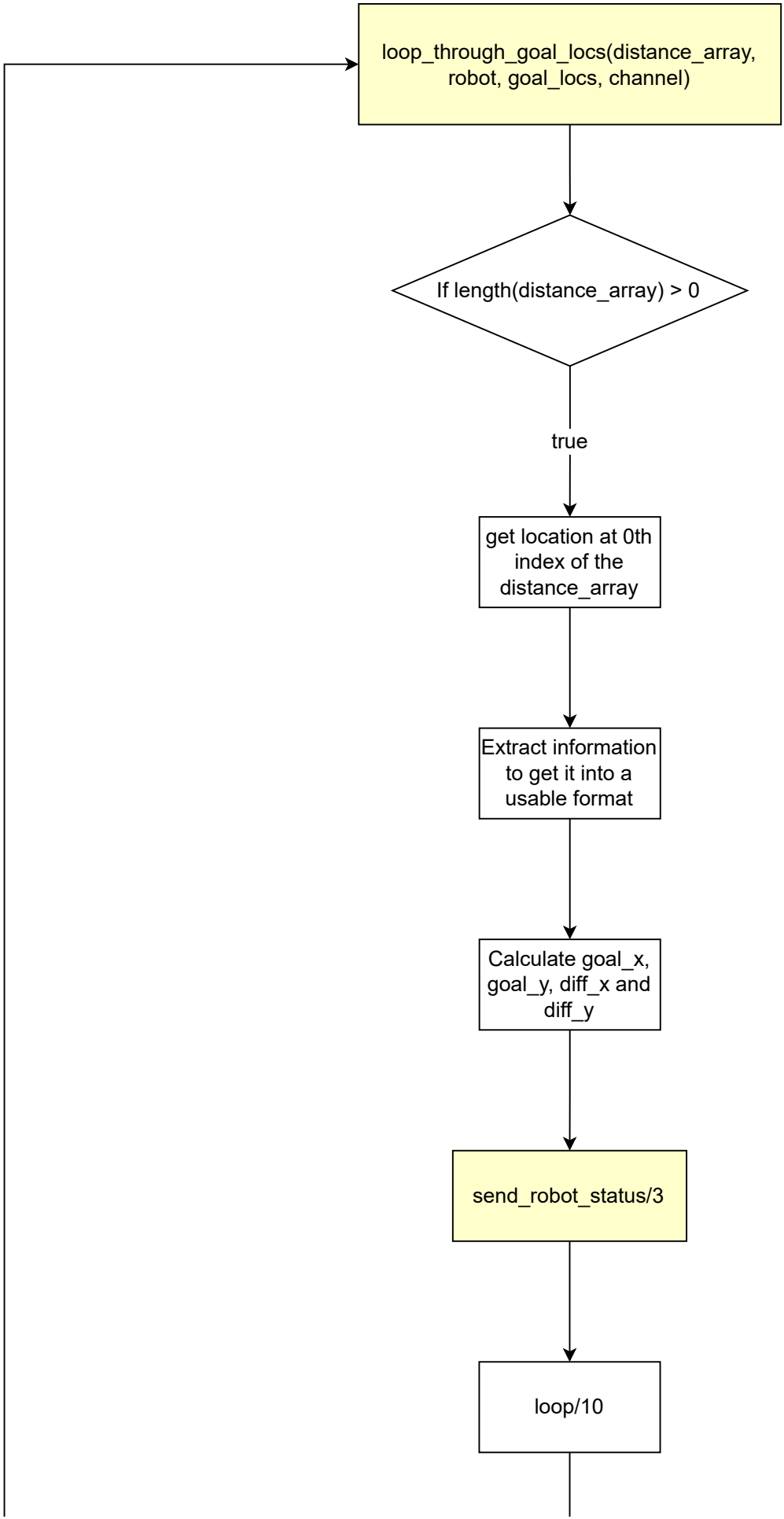

Server
(robot:update)
handle_in:
new_msg
goals_msg
coords_store_get
previous_store_get
goal_store_get
coords_store_update
previous_store_update
goal_store_update
goal_store_delete
event_msg
Client Robot A (Weeder)
User Socket
Client Robot B (Seeder)
User Socket
PubSub
Live View Page
handle_info({"update"})
handle_info({"update_obs"})
Agents

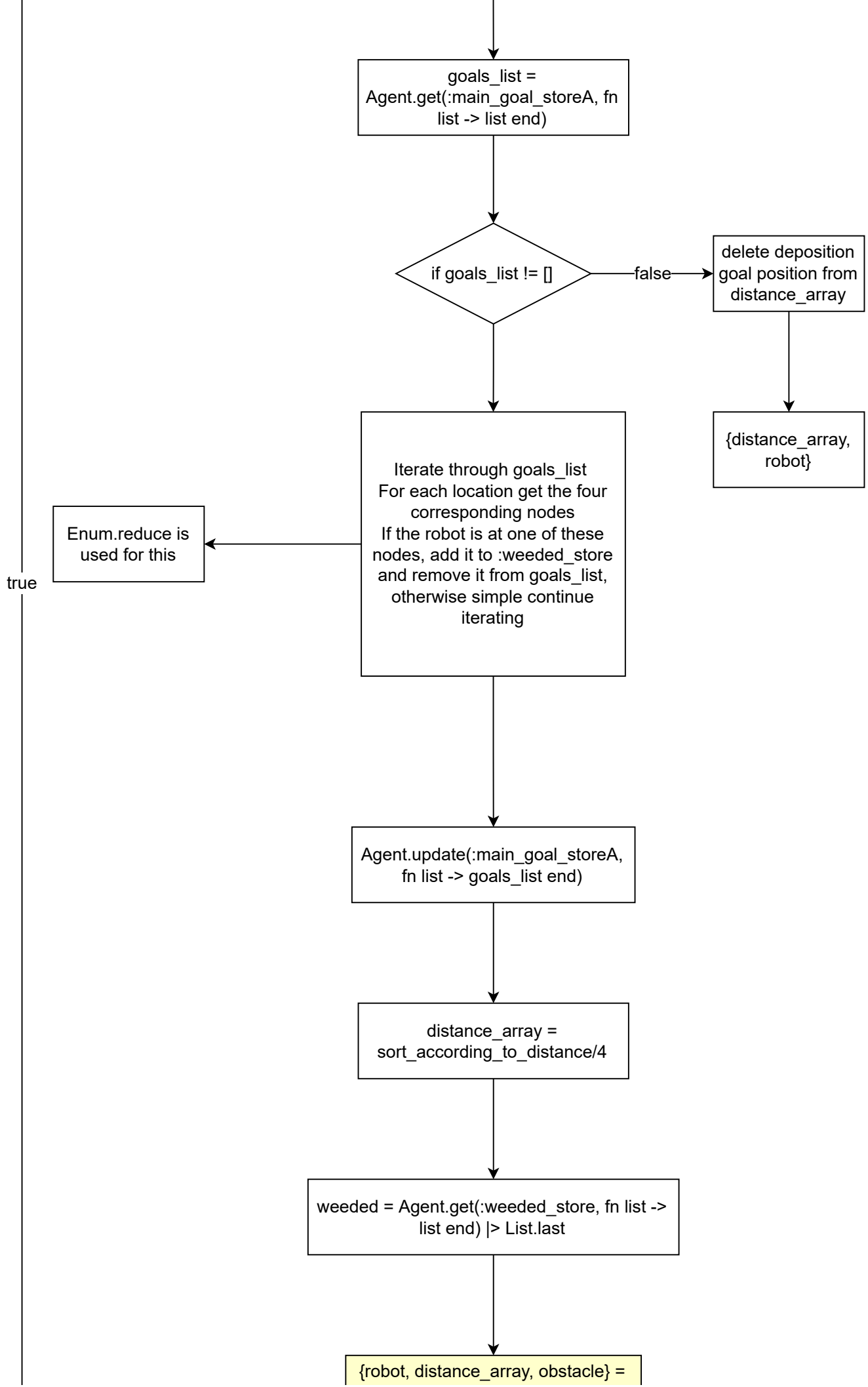
FWClientRobotA

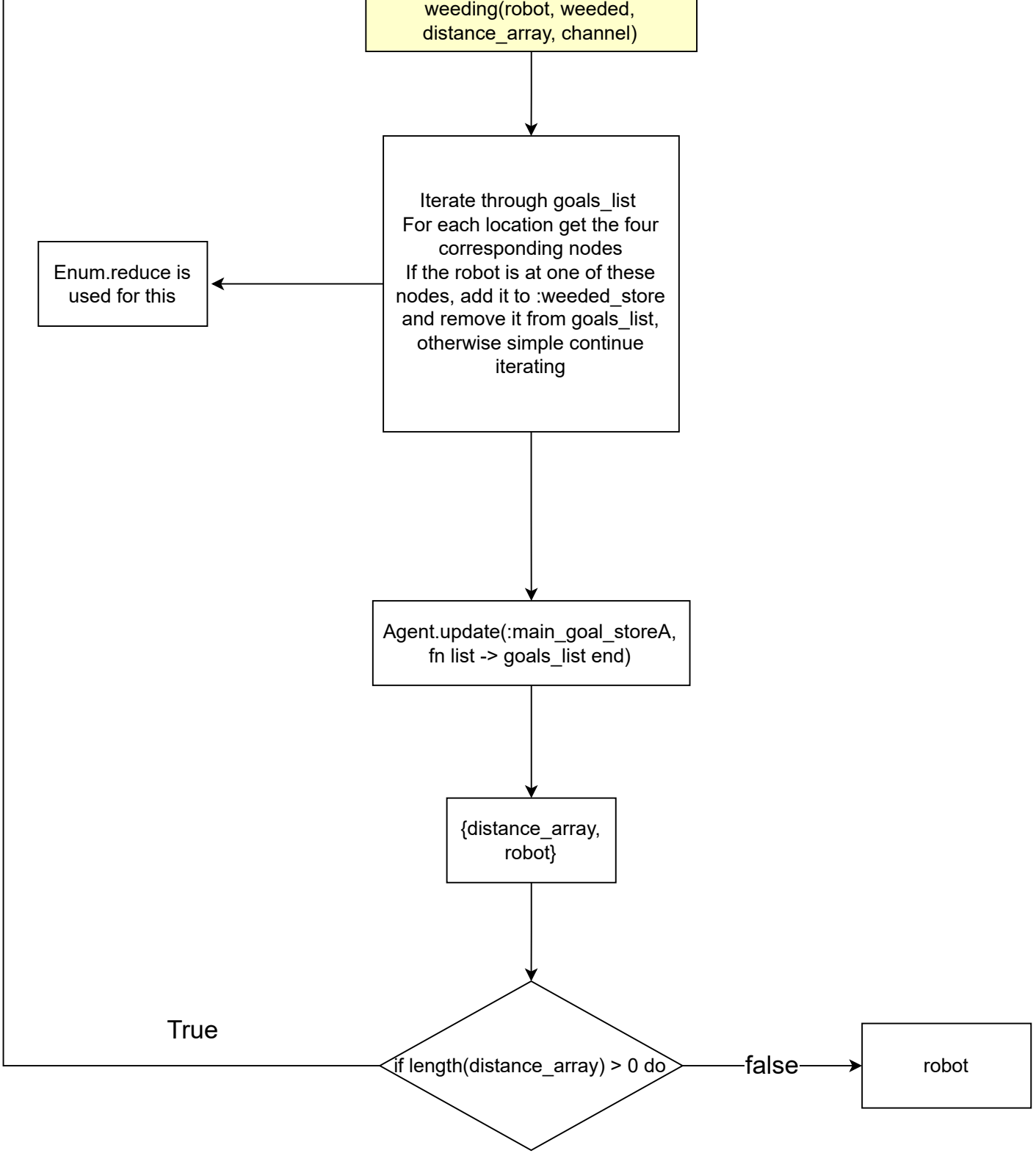


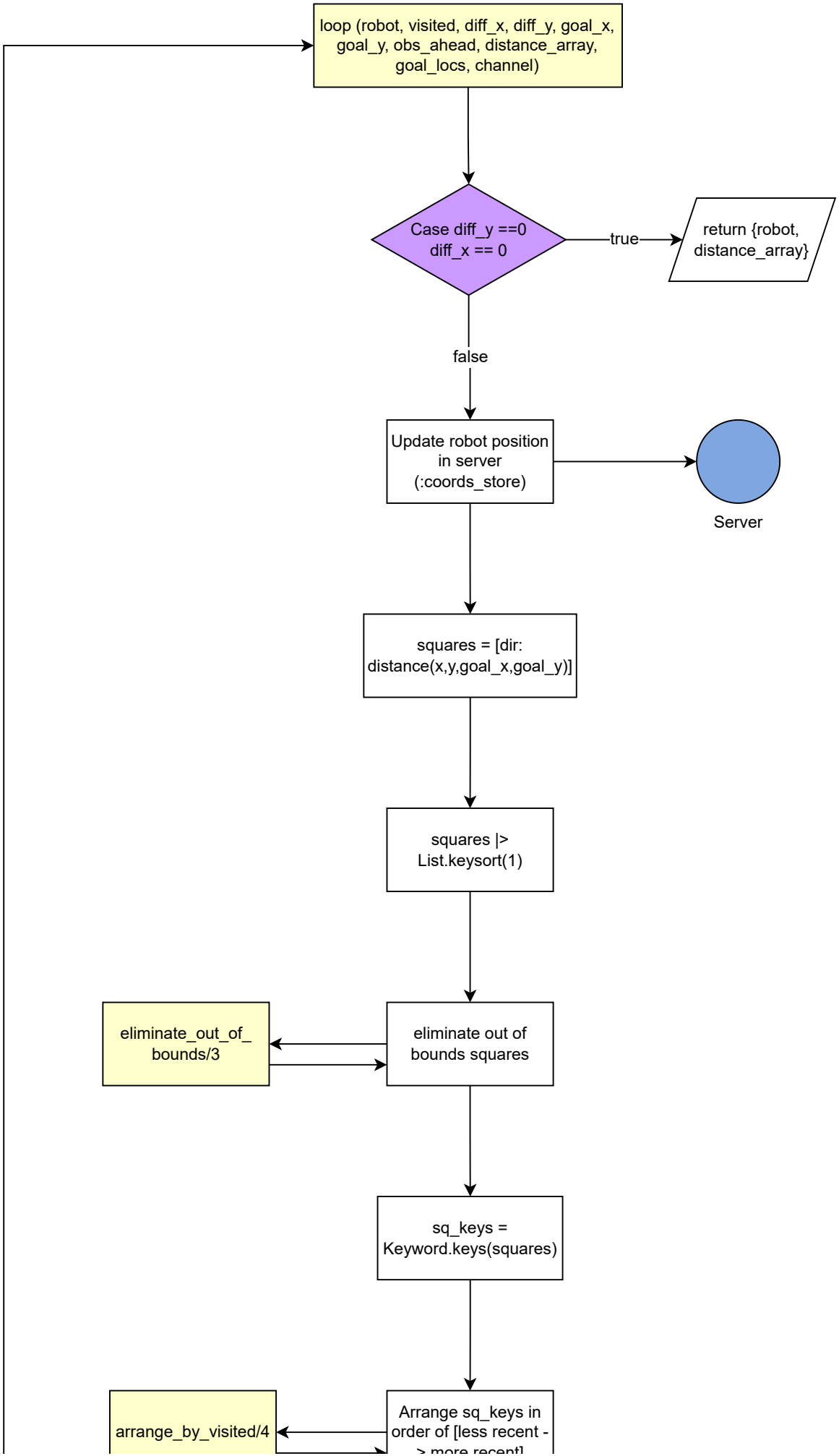

Convert the locations sent by the server into usable nodes by the robot for navigation
goal_locs = calculate_goals(robot, goals_string)
Go further in the navigation algorithm
stop/3
Send the final message that signals work completion by the robot
FWClientRobotA.PhoenixSocketClient.work_complete(channel)

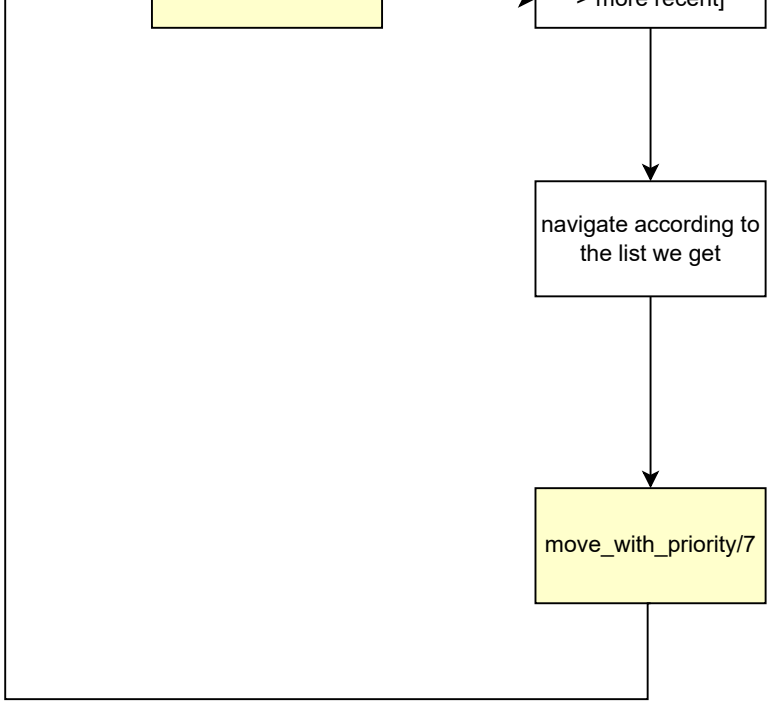

stop(robot, channel, goal_locs)
Sorts goal_locs according to their distance from the current position of the robot
distance_array = sort_according_to_distance/3
robot = loop_through_goal_locs/4
false
If length(distance_array) == 0
true
send_robot_status/3
Calculate closest deposition node to current position -> distance_array
loop_through_goal_locs/4

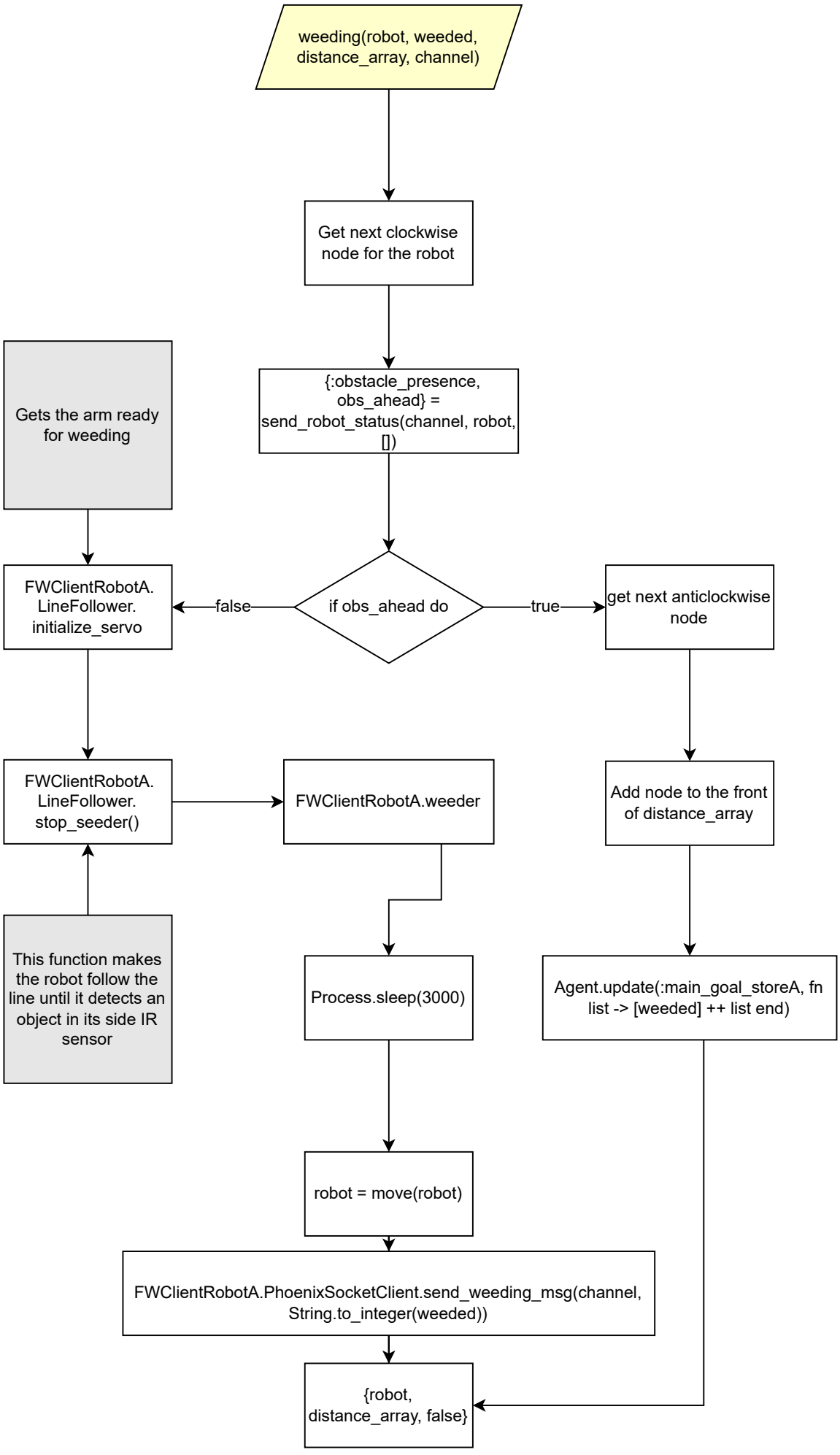

loop_through_goal_locs(distance_array, robot, goal_locs, channel)
If length(distance_array) > 0
true
get location at 0th index of the distance_array
Extract information to get it into a usable format
Calculate goal_x, goal_y, diff_x and diff_y
send_robot_status/3
loop/10

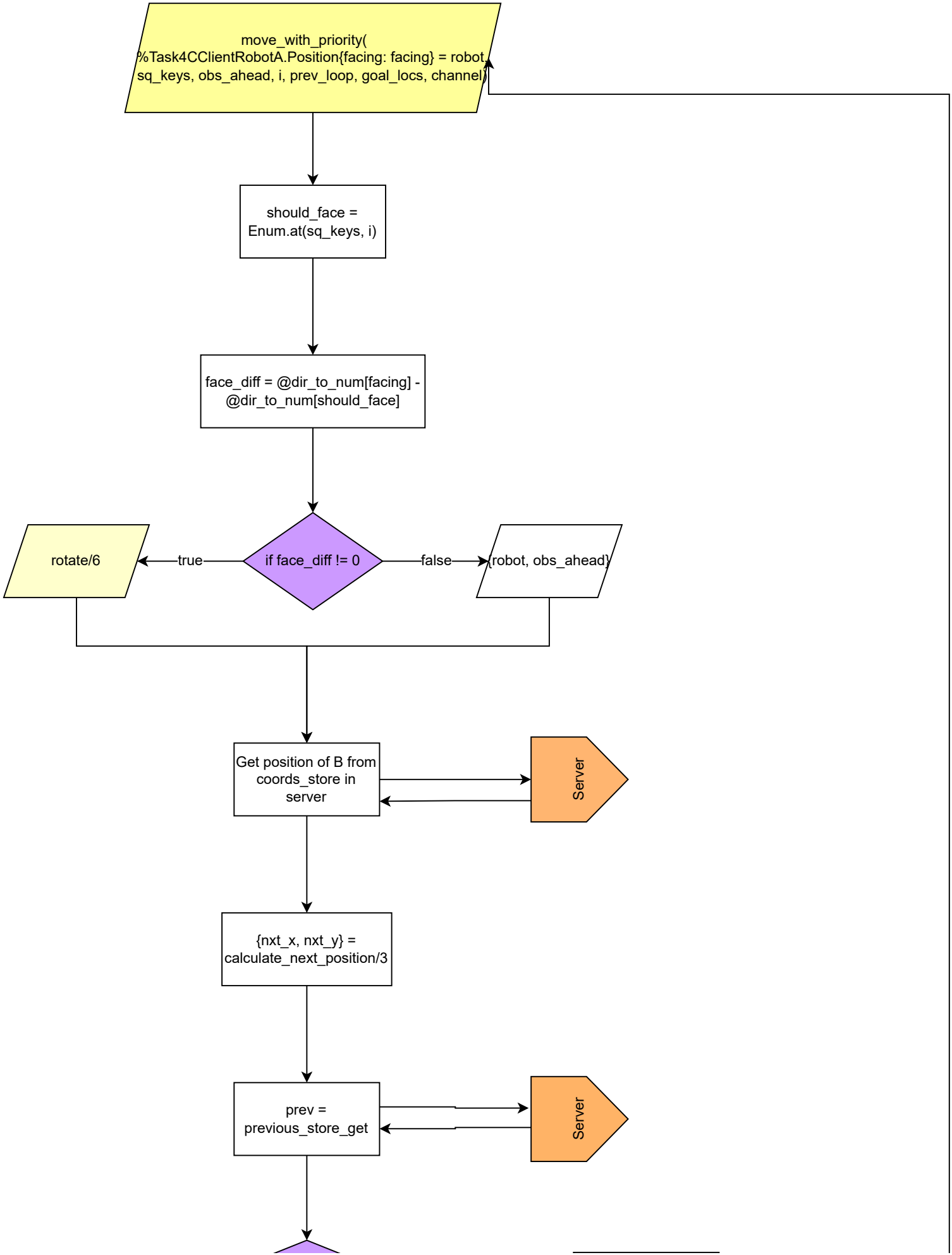

goals_list = Agent.get(:main_goal_storeA, fn list -> list end)
if goals_list != []
false
delete deposition goal position from distance_array
{distance_array, robot}
Iterate through goals_list For each location get the four corresponding nodes If the robot is at one of these nodes, add it to :weeded_store and remove it from goals_list, otherwise simple continue iterating
Enum.reduce is used for this
true
Agent.update(:main_goal_storeA, fn list -> goals_list end)
distance_array = sort_according_to_distance/4
weeded = Agent.get(:weeded_store, fn list -> list end) |> List.last
{robot, distance_array, obstacle} =

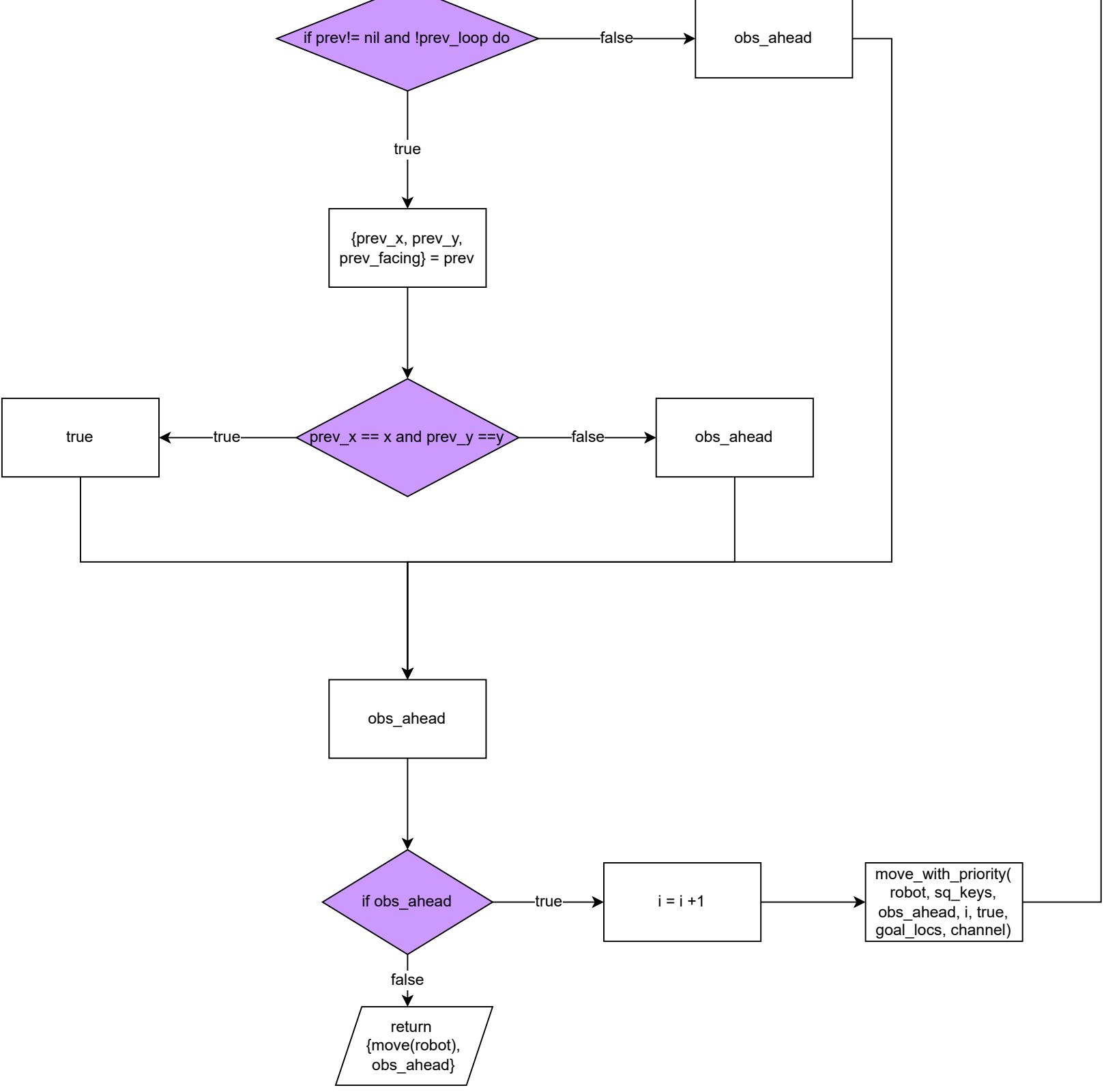

weeding(robot, weeded, distance_array, channel)
Iterate through goals_list For each location get the four corresponding nodes If the robot is at one of these nodes, add it to :weeded_store and remove it from goals_list, otherwise simple continue iterating
Enum.reduce is used for this
Agent.update(:main_goal_storeA, fn list -> goals_list end)
{distance_array, robot}
if length(distance_array) > 0 do
True
false
robot

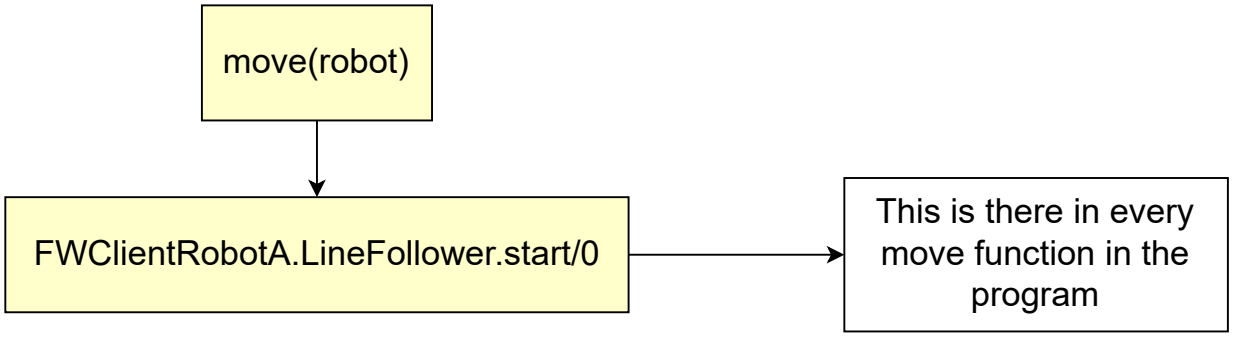

loop (robot, visited, diff_x, diff_y, goal_x, goal_y, obs_ahead, distance_array, goal_locs, channel)
Case diff_y ==0 diff_x == 0
true
return {robot, distance_array}
false
Update robot position in server (:coords_store)
Server
squares = [dir: distance(x,y,goal_x,goal_y)]
squares |> List.keysort(1)
eliminate_out_of_ bounds/3
eliminate out of bounds squares
sq_keys = Keyword.keys(squares)
arrange_by_visited/4
Arrange sq_keys in order of [less recent -

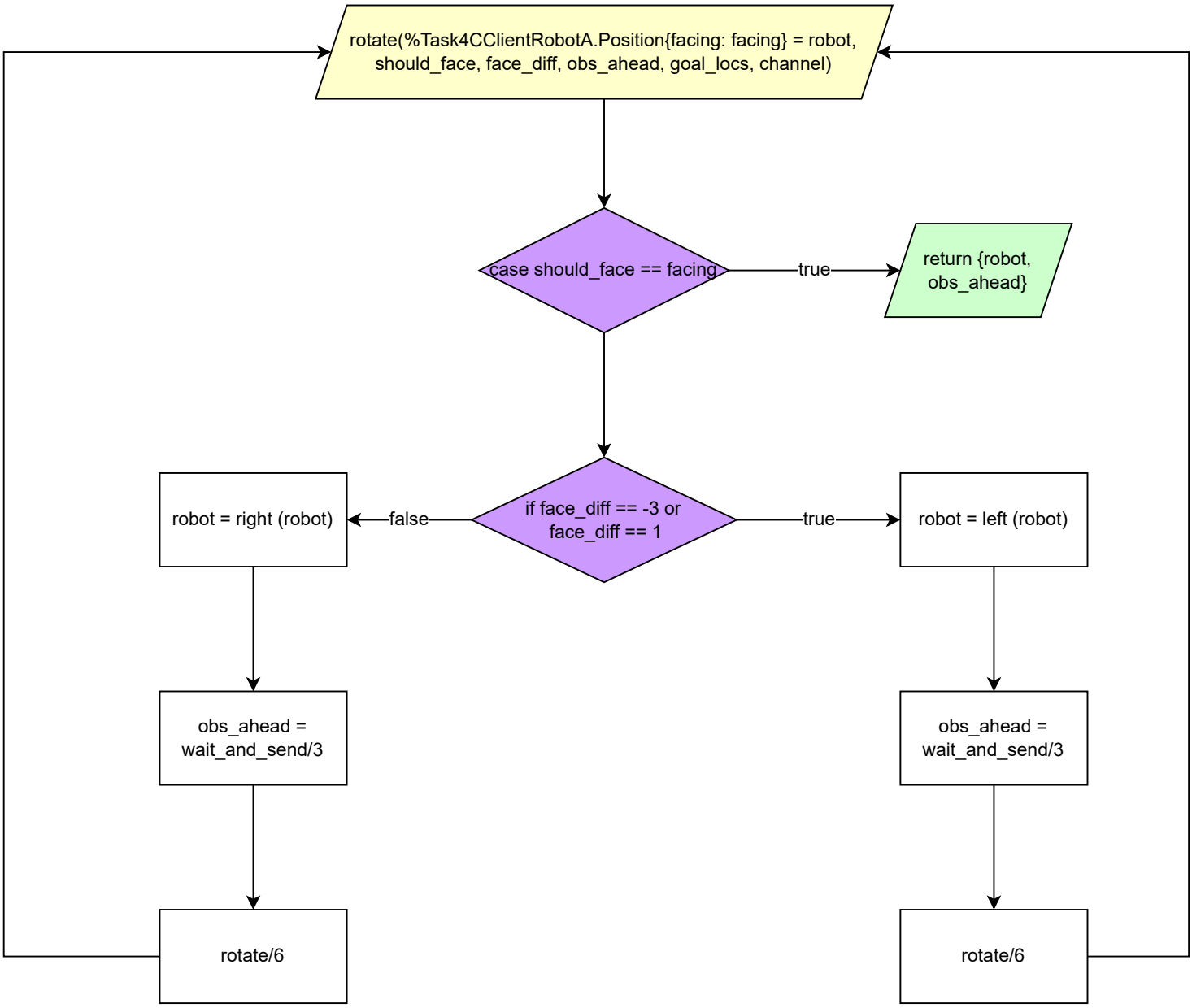

navigate according to the list we get
move_with_priority/7

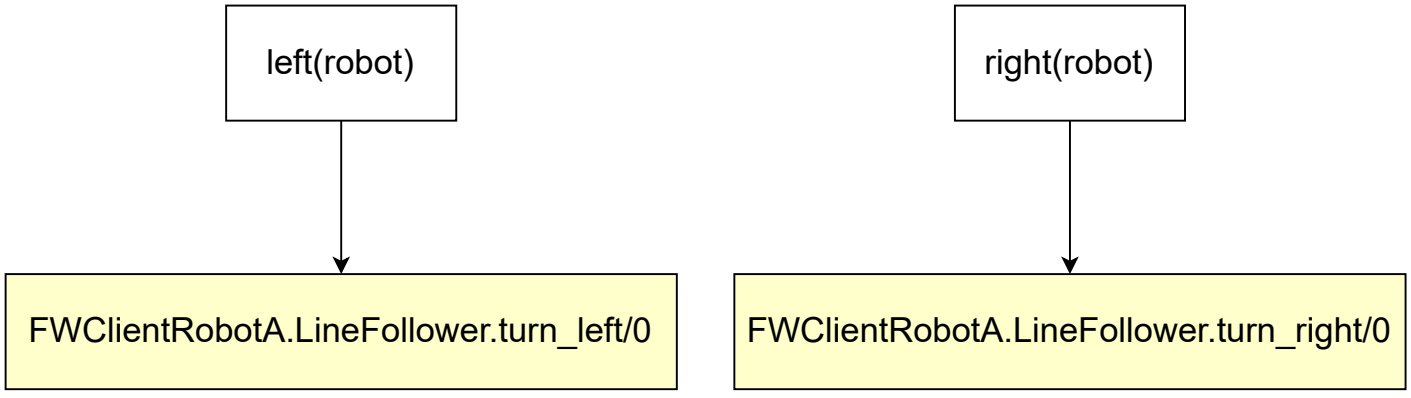

weeding(robot, weeded, distance_array, channel)
Get next clockwise node for the robot
{:obstacle_presence, obs_ahead} = send_robot_status(channel, robot, [])
Gets the arm ready for weeding
if obs_ahead do
false
true
FWClientRobotA. LineFollower. initialize_servo
get next anticlockwise node
FWClientRobotA. LineFollower. stop_seeder()
FWClientRobotA.weeder
Add node to the front of distance_array
This function makes the robot follow the line until it detects an object in its side IR sensor
Process.sleep(3000)
Agent.update(:main_goal_storeA, fn list -> [weeded] ++ list end)
robot = move(robot)
FWClientRobotA.PhoenixSocketClient.send_weeding_msg(channel, String.to_integer(weeded))
{robot, distance_array, false}

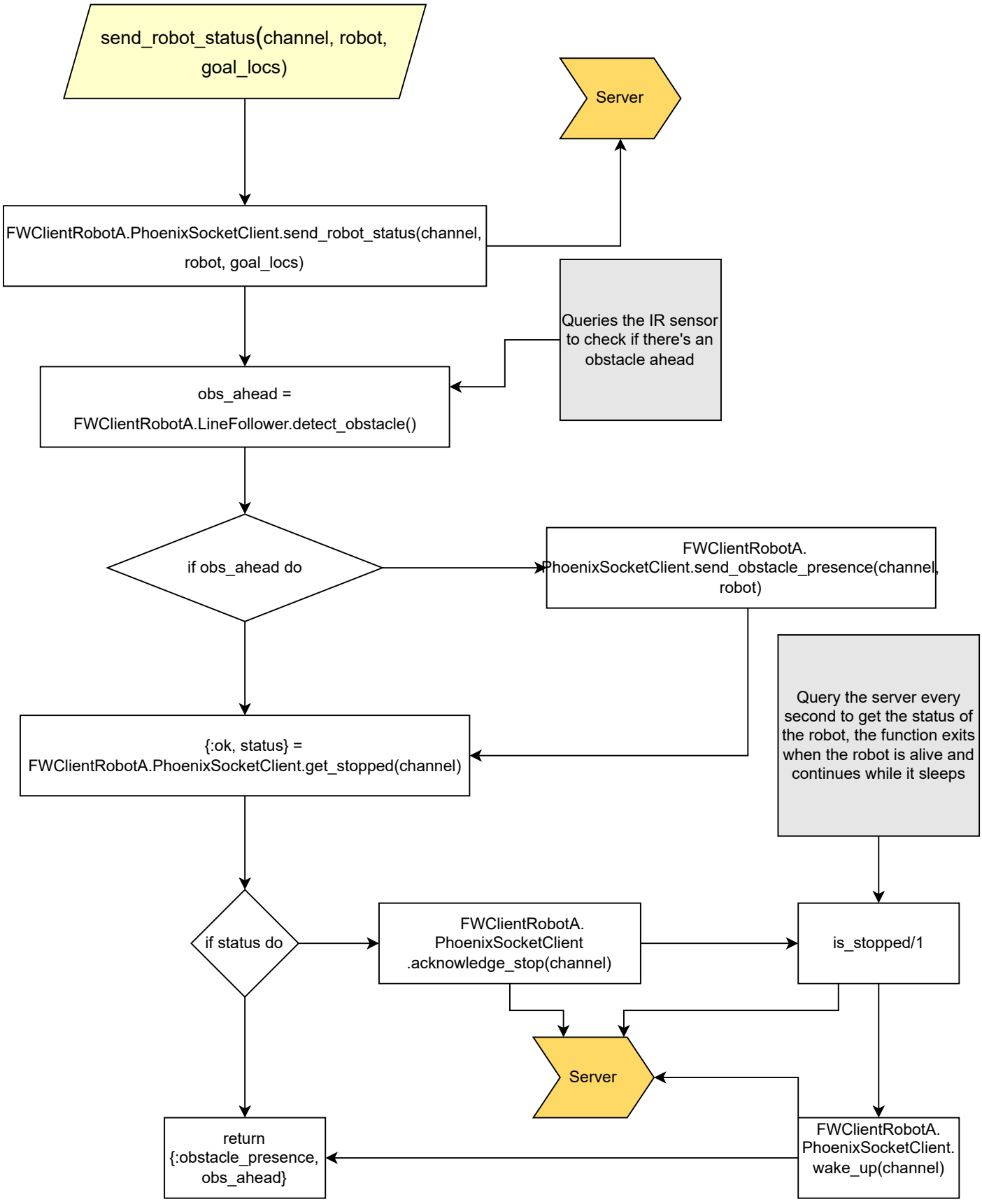

move_with_priority(
%Task4CClientRobotA.Position{facing: facing} = robot,
sq_keys, obs_ahead, i, prev_loop, goal_locs, channel)
should_face =
Enum.at(sq_keys, i)
face_diff = @dir_to_num[facing] -
@dir_to_num[should_face]
if face_diff != 0
true
rotate/6
false
{robot, obs_ahead}
Get position of B from
coords_store in
server
Server
{nxt_x, nxt_y} =
calculate_next_position/3
prev =
previous_store_get
Server


if prev!= nil and !prev_loop do
false
obs_ahead
true
{prev_x, prev_y, prev_facing} = prev
prev_x == x and prev_y ==y
true
true
false
obs_ahead
obs_ahead
if obs_ahead
true
i = i +1
move_with_priority(robot, sq_keys, obs_ahead, i, true, goal_locs, channel)
false
return {move(robot), obs_ahead}


move(robot)
FWClientRobotA.LineFollower.start/0
This is there in every move function in the program


rotate(%Task4CClientRobotA.Position{facing: facing} = robot, should_face, face_diff, obs_ahead, goal_locs, channel)
case should_face == facing
true
return {robot, obs_ahead}
if face_diff == -3 or face_diff == 1
false
robot = right (robot)
true
robot = left (robot)
obs_ahead = wait_and_send/3
obs_ahead = wait_and_send/3
rotate/6
rotate/6


left(robot)
right(robot)
FWClientRobotA.LineFollower.turn_left/0
FWClientRobotA.LineFollower.turn_right/0


send_robot_status(channel, robot, goal_locs)
Server
FWClientRobotA.PhoenixSocketClient.send_robot_status(channel, robot, goal_locs)
Queries the IR sensor to check if there's an obstacle ahead
obs_ahead = FWClientRobotA.LineFollower.detect_obstacle()
if obs_ahead do
FWClientRobotA. PhoenixSocketClient.send_obstacle_presence(channel, robot)
Query the server every second to get the status of the robot, the function exits when the robot is alive and continues while it sleeps
{:ok, status} = FWClientRobotA.PhoenixSocketClient.get_stopped(channel)
if status do
FWClientRobotA. PhoenixSocketClient .acknowledge_stop(channel)
is_stopped/1
Server
return {:obstacle_presence, obs_ahead}
FWClientRobotA. PhoenixSocketClient. wake_up(channel)