Collision Avoidance on Drones

Distributed Systems - Project 2023

Luca Rosso, Andrea Giuseppe Zarola
DMIF, University of Udine, Italy

Version 0.1, March 15, 2023

Abstract

The aim of this project is the analysis and the implementation of a distributed solution for collision avoidance between drones in a 2 dimensional space.

The problem can be divided in 3 main subproblems: drone synchronization, collision detection and priority agreement.

Note that drones are autonomous systems without any external air traffic control and can send messages to other drones using wireless technology in order to communicate. The system must be also robust to failures especially those regarding drones and must always guarantee that a delivery must eventually be completed. The project implements a simulator for the environment, business logic and a Rest API that exposes the simulation state for any other future UI implementations.

Contents

Chapter 1

Introduction

In this introductive chapter we are going to describe our problem in detail and provide a list of the requirements that must be satisfied by the system.

1.1 Definition of the problem

Description of the application

A big shipping company desires to start delivering packages using drones. The company is worried by some possibles collisions between the drones when transporting the merch. The drones have the capacity to transport only one delivery per trip and they all fly at the same height with a constant speed, for this specific reason we need a solution for collision avoidance. We assume that all the drones are in a connected network sending messages with some kind of communication standard (Wifi, Bluetooth, etc..) and can communicate to all the other drones in the network. We assume that for each new delivery a new drone will be used. We assume also that at the end of each delivery the drone will be left on the ground.

We assume that drones can fail during a delivery, when that is the case a new delivery will be created from the point of the failure to the original delivery point and another drone will pick up the delivery. When a drone fails it does not represent a threat of collision for our system anymore because we assume that a failed node will land safely to the ground waiting for a new drone to arrive. We assume that the drone, after the delivery is completed, will send a message to the Rest node for updating the DB. We assume that a drone cannot be malicious, in fact once the route is established a drone cannot change it once in flight to try to collide with other drones.

For the representation of the space we use a 2D cartesian plan, the deliveries are couples $((X_1, Y_1); (X_2, Y_2))$ and we represent the drones as squares of length $l >= 1$. We cannot have two drones flying to close to each other because they can collide due to their size. The routes are added dynamically.

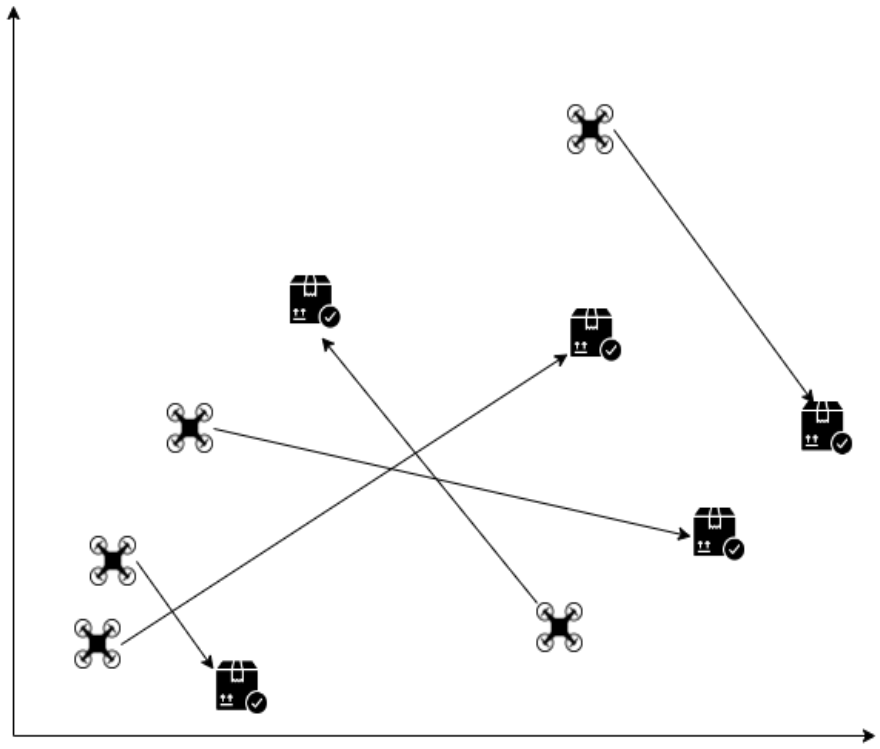

Figure 1.1: Example of a possible situation

Overall structure of the implementation

We decided to divide the system into 4 main subsystems:

- **UI Control Service**: The main source of UI for the user.
- **Distributed DBMS**: Stores the data from all the deliveries.
- **Rest API**: Used for providing primitives for the drones and the drone hub, for client interaction and data persistence.
- **Delivery Service**: The drone subsystem, composed by a drone hub process and multiple drone processes.

Implemented transparencies

Our system satisfy these forms of transparency:

- **Location transparency**: there is no difference in the way the resources are accessed. Using the Rest API we have identical operations to access the resources.

- **Concurrency transparency**: Several drone processes are alive at the same time not interfering with the routes of other drones.
- **Failure transparency**: A drone that fails a delivery will eventually in the future be replaced by a new drone that will complete the pending failed delivery.
- **Scaling transparency**: The system can scale in the size of the drones and wrt. the distributed database.

1.2 Requirements

The main requirements to be met are:

- **Fairness**: The fairness of the system wrt. delivery completion directly depend by the priority policy chosen by the system. Assuming that the system uses a fair policy, than also delivery completion is fair.
- **Fault tolerance**: Every drone that experience a failure will be replaced with a new recovery drone by the system.
- **Starvation(No drones waiting forever)**: Every drone that enters the system will eventually takeoff and complete the delivery.
- **Deadlock(No deadlock between drones)**: It is not possible that two or more drones wait for each other.

Chapter 2

Analysis

In this section we discuss the requirements that the system must satisfy. Requirements are divided between functional and non functional.

2.1 Functional requirements

- The system allows the insertion of a new delivery with any starting and final point on the ground.
- The system notifies a signal on the terminal every time a requested delivery is completed.
- The system notifies a warning on the terminal every time one of the drones currently flying fall down.
- The starting and final points of a new delivery must be inserted in the system through a couple of 2D points.
- Each delivery is identified by a unique identifier which allows users to control the current status of the delivery.
- All the drones that start their deliveries must follow the rule that collisions must be avoided.

2.2 Non functional requirements

- The coordinates of the starting and final point of each delivery must always be positive, and must be less or equal to the size of the ground in which the system is deployed.
- Each drone completes its delivery following a linear path and with a constant speed.
- Each drone flies at a prefixed height that is reached at the start of the delivery with a constant speed.

- Every time a new delivery will be created a new drone will be used.
- If one of the drones falls during the travel, a new drone will be spawned from the system in order to complete the delivery from the point where the drone fell.
- If multiple drones at the same time are not in collision with any other, then the two drones will start their deliveries simultaneously.
- The system guarantees a correct agreement between the drones on the ground, in a way that it is not possible that two drones enter in a collision during their paths. The requirement must be satisfied through a distributed algorithm of collision avoidance in which the drones themselves are responsible to compute the collisions.
- The computation of a potential collision between the paths of two different drones must consider also the body size of the drones.
- All the deliveries eventually will be completed. It is not possible that a drone will wait forever to complete its delivery.
- It cannot happen that two or more drones wait for each other during the agreement phase in order to start their deliveries.
- If a drone falls during the completion of its delivery, the new drone that the system spawns will complete the delivery in the same final point where the fallen drone should have arrived, and starting from the point in which the first drone fell.
- The order in which the drones must start their delivery must be decided by the drones themselves (in agreement) without any external help.
- If two or more deliveries have their respective paths intersecting, the system will try to maximize the number of flying drones in according to a specific policy.

Chapter 3

Project

This chapter is devoted to the description of the general architectures, and specific algorithms.

3.1 Logical architecture

The logical architecture is composed by the following main components:

- **Rest API**: A node that provides informations to the drones about the network and at the same time can send queries to the DB and respond to the UI Control Service.
- **UI Control Service**: A shell that can be used by a user to create new deliveries, retrieve information about them, monitor deliveries, and kill drones that are dealing with a specific delivery, by interacting with the Rest API.
- **Distributed DBMS**: An instance of Mnesia DBMS paired with the web service instance (it can scale with multiple DBMS containers and multiple Rest API instances).
- **Drone Hub**: A process that acts as a supervisor of the drones. It has the ability to:
  - Kill existing drones.
  - Spawn new drones.
  - It is informed when a drone fails.
- **Drone**: A process that is an abstraction/simulation of a real life drone.

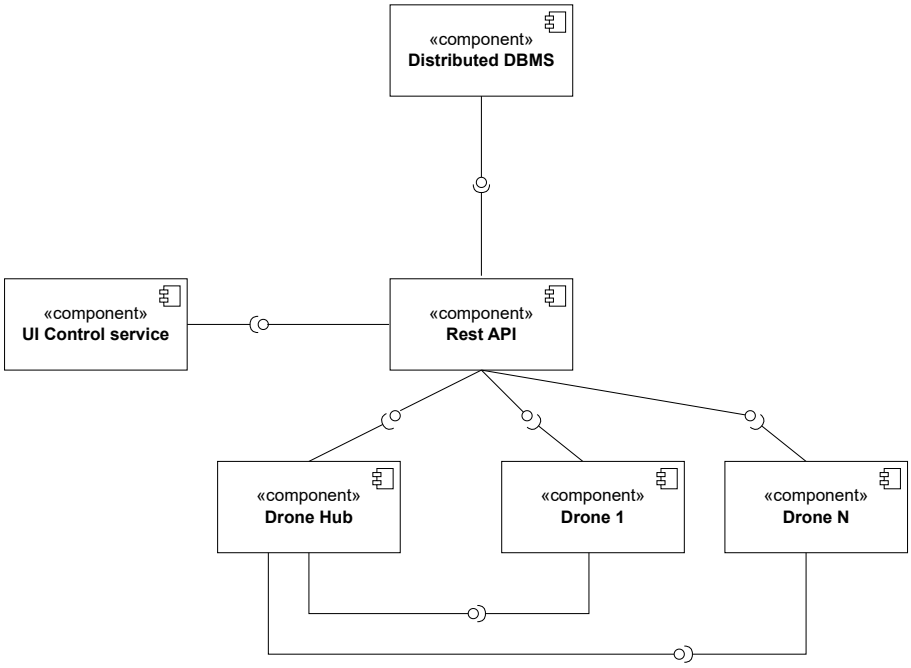

Figure 3.1: Component Diagram for the system

3.2 Protocols and algorithms

Synchronization between drones

At the start, each drone knows only its path, but does not know if its path is in collision with the path of others and in this case who are these drones in collision with it.

Solution

Each drone will send a request of *computeCollision*, with its route (the couple of starting and final points) as parameter, to any other drone with a lesser id.

At the same time, any drone can receive requests of *computeCollision* from all other drones with a greater id.

When a drone receive a request of *computeCollision*, it computes if there's a collision or not with its route: if a collision is found, then it will send back a message of *collisionDetected* otherwise it will send back a message of *collisionNotDetected*.

When a drone D knows the list of drones colliding with him (i.e. it has processed all the received requests of *computeCollision* and it has received the response for each of the requests it sent) D sends a tuple containing the drones colliding with him and its state (if it is flying or on the ground) to all the nodes who are in collision with D. Moreover, the state comprises also the value of

a counter *notify_count* which indicates the number of times a drone has given precedence to other drones (at the start in each drone this counter is 0).

In this way, every drone D maintains a table of collisions which says who are the colliding drones for each drone in which D is in collision, and it knows also the state of each of its colliding drones.

An example of interaction between three different drones can be seen in Figure 3.2.

In this case the drone D_1 has its path in collision with only the path of D_2, D_2 has its path in collision with both the path of D_1 and D_3, and D_3 has its path in collision only with the path of D_2.

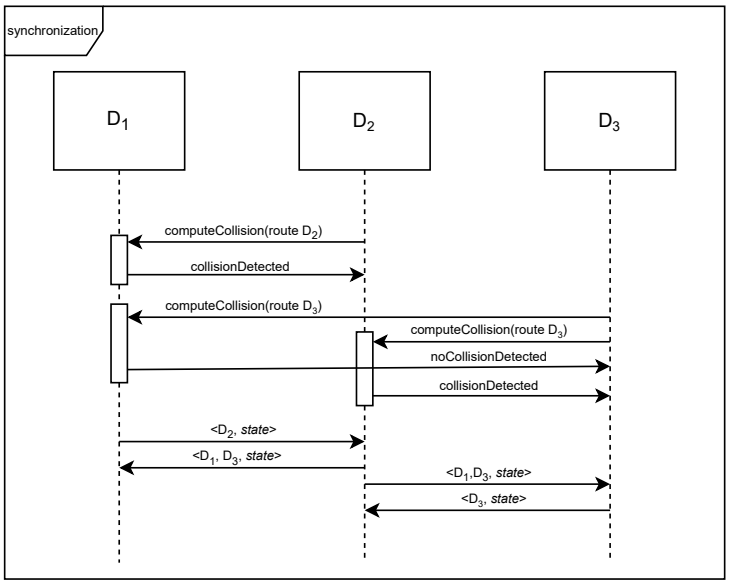

Figure 3.2: Sequence diagram of the synchronization phase

A drone can always receive requests of *computeCollision* both to know new drones that joined the network and collide with it and also to allow the new drones to synchronize with all the other drones.

Agreement between drones

When a drone is synchronized with all the other drones, it knows who is in collision with it but doesn't know who has the priority to fly or who must wait to fly.

Solution

It's trivial to state that when a drone does not collide with any other, it can start immediately to fly.

The situation starts to become much more critical when drone's paths collide with other drone's path: solution is to make that drones that know to

collide with others participate in an agreement algorithm in order to decide who can start to fly and who must wait for its turn.

When a drone D has its route in collision with the route's of at least another drone, thanks to the synchronization phase, this means that D knows also who are the colliding drones for each of its colliding drones. Along with the state of each colliding drone, this information can be seen in a tabular way and allow a drone D to decide, at least for its personal behavior, who are the next drones that have the priority to start to fly.

So, in order to choose who has the priority, each drone D must order the list of its colliding drones following three different rules in this order:

- if on the table, there are drones with a state equals to "flying" they must be put at the start of the order considering the id in ascending order;
- if on the table, there are drones with a value for *notify_count* greater than a fixed threshold (which value is global in the whole system) then these drones must follow in the ordering considering the id in an ascending order;
- the last drones that must appear in the order are the remaining drones in the table, ordered first by the number of collisions with other drones (in ascending order) and then by the id (in ascending order).

If the first drone in the order is exactly D, this means that D has the priority to start to fly and needs only to wait the confirmation by the drones that have their paths in collision with path of D (through a *notify* message from each of them). When a drone receives a *notify* message from all the drones that have their path in collision with the D's path, then D can start to fly.

When the first drone in the order is not D, then this means that D must give priority to start to fly to others. In particular, D has to scan the order from the left to right and give confirmation of priority (through a *notify* message) to each node if and only if the current scanned drone does not have its path in collision with the path of the drones to which it has already sent confirmation in the current round of the algorithm.

For each of the drones to which D sent a *notify* message, then D must wait for the respective ack message that will be sent by each drone only when it is sure that a collision with D cannot more happen (i.e. the drone passes the point of intersection with the path of D). In this case, a drone D must wait to receive the ack message from all the drones to which it sent *notify*, before D can start again to consider who has now the priority. Moreover, D must add each of the drones to which it sent notify in its *notify_count*.

At the end, for a process D it's important to remove the drones for which it has received the ack from the table.

Moreover, before D restarts the algorithm it needs to update its entry in the table of each drone that is yet in collision with D: in particular, D must send an *update_table* message to each of them which will contain the remaining drones for which D is colliding and the new value for the *notify_count*.

Following this algorithm, it's possible that while a drone D is waiting to receive ack messages, a new request of *computeCollision* is received from a new drone that joined the network: in this case, if D found that there's a collision with the path of the new drone, after synchronizing with it, then can simply check if it can give priority to this new drone according to the priority that have been already given to the other drones in the current round.

An example of an interaction between three different drones can be seen in Figure 3.3.

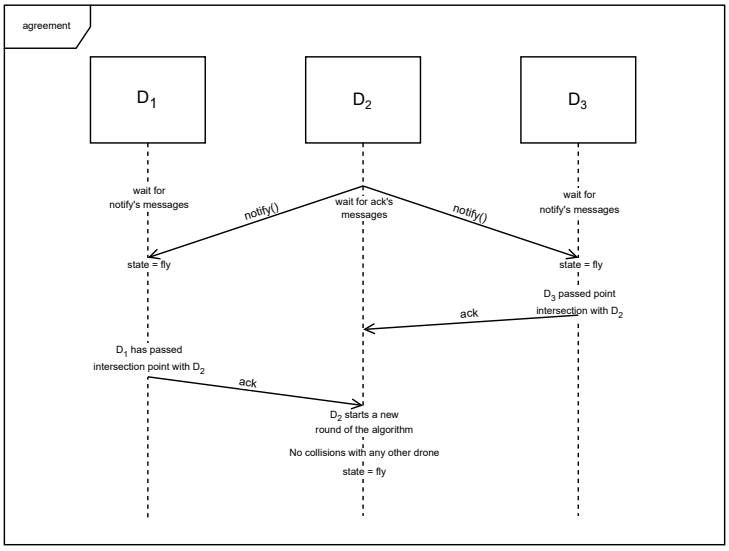

Figure 3.3: Sequence diagram of the agreement phase

Following the example in Figure 3.2, D_1's path collide only with the path of D_2 and the same happen for D_3 with D_2. So, from their perspectives D_1 and D_3 know that they must wait for *notify*'s messages, while since D_2 is the drone whose path collides with the two others knows that it must give priority to D_1 and D_3. Then, when D_2 has received ack's messages from both D_1 and D_2 it knows that it hasn't any more collisions with other drones, so it can start to fly also.

Fault tolerance

The system must be resilient to the failure of single drones.

Solution

A drone can fail when it is on the ground during the agreement phase or while it is flying. In case of failure when it is still on the ground, we must distinguish two different situations:

- the drone has failed after sending some *notify* messages and so, while it was waiting the respective ack's messages: in this case, since each drone

is monitored by the drones hub, after the failure a new drone (a recovery drone) will be spawned from the hub but with the same route and the id of the failed drone. So, as with any new drone when it joins the network, the recovery drone must synchronize with all the other drones in the network. Since the synchronization phase also involves the other drones, any drone that has already received a *notify* message from the failed drone and now receives a synchronization message from the recovery drone must behave according two possible situations:

  - if the drone has already started to fly, then the recovery drone will already know this information from the state and must simply send a *notify* message following the rules described in the 3.2 section;
  - if the drone hasn't already started to fly (because it is still waiting for *notify* messages from other drones), then the drone must simply ignore the *notify* message from the failed drone (if it already received it) and wait for a *notify* message from the recovery drone.

- the drone has failed while waiting some *notify* messages from other drones: the drones hub will spawn a new drone (a recovery drone) with the same route and id of the failed node. After the synchronization phase of the recovery drone with all the other drones in the network, the drones that have already sent a *notify* message to the failed drone must simply ignore the failed drone from the list of drones from which they are waiting ack messages and treat the recovery drone as a new drone.

In case of failure on a drone which was flying(i.e. with the state = "flying"), the solution is that any drone that is flying must send updates about its current position to the Rest API every time it travels for a distance equal to the minimum distance with which a drone can travel in a second. Moreover, this update must be done also every time that a drone sends an ack message to another drone.

In this way, when a drone fails, the drones hub will spawn a new drone (a recovery drone) with the same id, the same final point, a "pending" state, a *notify_count* equals to 0 but with a starting point equals to the last position that was updated on the Rest API.

So, after the recovery drone will complete the synchronization phase with the other drones in the network, the other drones can be in two different situations:

- for the nodes that have already received the ack message, there aren't problems since the starting point of the new drone will be after the potential intersection point;

- for the nodes that haven't already received the ack message, since they were also involved in the synchronization phase, they will simply remove the recovery drone from the drones from which they are waiting the ack message.

Scalability

The system must have the ability to scale with respect to the load (both client and drones).

Solution

Regarding the drones, the simulation of each of them can be deployed in different containers. In our case, we will deploy all these processes locally on the same machine, but in a real case each of the drone's containers can be deployed on the machine that controls the drone.

For the UI Control Services, and the Rest API, it's possible to replicate them on different machines and then use a load balancer to manage the loads and also to improve robustness to failures.

Finally, regarding the distributed DBMS (Mnesia), we will use it locally on a single node but in a real case it can be easily distributed on multiple nodes.

3.3 Physical architecture and deployment

As can be seen in Figure 3.4, the system is deployed using Docker containers. We use separate containers for the UI control service, the Rest API, and the Distributed DBMS. In particular, for our needs we plan to deploy the distributed DBMS only on a single node but in a real case the distributed DBMS should be deployed on multiple nodes: this means to isolate each node from the others in different containers. Regarding the single drone processes, we isolate each of them in different docker containers: this allows us to not have a single point of failure between them. Furthermore, also if the drone's hub fails, the already active drones could continue to live, since the link between the hub and the drones is unidirectional.

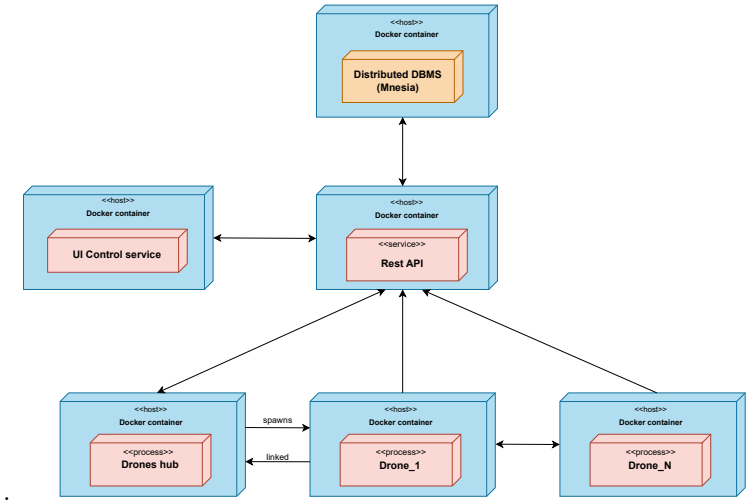

Figure 3.4: Deployment diagram of the system

3.4 Development plan

We have decided to split the development plan into 4 different tiers.

- **Tier 1**: The system gives the user the opportunity to insert new deliveries from the UI Control Service.
- **Tier 2**: The system is able to guarantee the collision detection between drones, and allows to insert new deliveries from the UI Control Service. No fault tolerance is ensured at this level.
- **Tier 3**: The system is able to guarantee collision detection, fault tolerance for deliveries, and allows to insert new deliveries from the UI Control Service.
- **Tier 4**: The system is able to guarantee collision detection, fault tolerance, delivery data persistence, and allows to insert and monitor new deliveries from the UI Control Service.

Chapter 4

Implementation

In this chapter we will explain how the system was implemented, so we will describe technologies/languages, data structures, patterns and best practices used.

4.1 Language

We choose to use Erlang as language of implementation since we think it's the most adapted for our case. Erlang is a functional language that allows multiple lightweight processes that can communicate asynchronously through the use of message passing.

4.2 Data Structure

We used a custom data structure for the Delivery that is constructed as follows:

- **Id**: Unique identifier of the delivery.
- **Pid**: Current Pid of the drone making the delivery.
- **State**: State of the delivery (pending, flying, completed).
- **Start_X**: Coordinate X of the starting position.
- **Start_Y**: Coordinate Y of the starting position.
- **Current_X**: Last known coordinate X reached by the drone.
- **Current_Y**: Last known coordinate Y reached by the drone.
- **End_X**: Coordinate X of the ending position.
- **End_Y**: Coordinate Y of the ending position.
- **Fallen**: Logs failure times for the drone.

The following is an example of a completed delivery stored in Mnesia.

```
#{
    <<"current_x">> => 50.6,
    <<"current_y">> => 67.9,
    <<"end_x">> => 50.6,
    <<"end_y">> => 67.9,
    <<"fallen">> => " ",
    <<"id">> => 0,
    <<"pid">> => <binary representation>,
    <<"start_x">> => 40.0,
    <<"start_y">> => 30.0,
    <<"state">> => <<"completed">>
}
```

In particular, we chose to store the binary representation of the Pid because, when there are processes that communicate remotely on different nodes, the direct use of the standard representation can be a problem since a process on a node may not be able to resolve the Pid of a process on a different node. In particular, before a process on a node starts to communicate to another, it first must decode the binary representation to the standard representation.

4.3 Implementations choices

Rest API

For the Rest API we decided to use the Cowboy Framework because it was the most widespread and supported Framework for Erlang to build Restful API. We also decided to use JSON as a standard interchange format using the Jiffy library. The main reason for this choice is that JSON is almost a standard de facto and it is also language independent and very versatile.

Database

For the Database layer we decided to use Mnesia for these main reasons:

- The ability to scale on multiple nodes and so to avoid a SPOF in the system.
- It is the most used DBMS for Erlang applications.

UI Control Service

For the main UI module of the system we decided to use an Erlang shell that wraps the following calls to the Rest API:

- `terminal:create_delivery(Start_X,Start_Y,End_X,End_y)`: It can be used to insert a new Delivery. Its input parameters are of type float and it returns the Id of the newly inserted delivery.

- `terminal:kill_delivery(Delivery_id)`: It can be used to simulate a crash of the drone dealing with a delivery. It returns as output the information whether the drone has been killed, or not if the delivery for the input Id hasn't been found.
- `terminal:watch_delivery(Delivery_id)` : It can be used to monitor a certain delivery throughout its journey and be notified every time there's a state change or if the drone dealing with the delivery falls.
- `terminal:get_delivery(Delivery_id)`: It can be used to retrieve information about a certain delivery.
- `terminal:create_and_monitor(Start_X,Start_Y,End_X,End_y)`: It can be used to create a new Delivery, but moreover it also logs on the shell all the state transitions for that specific delivery, and every time there's a failure on the drone that is dealing with it.

Drone hub

The drone hub serves the role of supervisor for all the drones. It has the main goals of:

- Notify the Rest API if a node has a failure
- Spawn a drone every time a new delivery is inserted in the system
- Spawn a recovery drone every time the drone that is currently dealing with a certain delivery falls

Moreover, it also sets the following environment variables for the simulation:

- Velocity: represents the velocity to which drones fly and it is expressed in m/s.
- Drone size: represents the size of bounding box used to represent physically each drone.
- Drone size: represents the size of bounding box used to represent physically each drone.
- Notify threshold: represents the threshold that is used from each drone to understand when it can get priority against the others independently from the number of collisions that a drone has.

All these attributes cannot change during the simulation, but can easily be edited at compile time.

Synchronization algorithm

For the synchronization problem we decided to use a distributed algorithm that exchange four kind of messages:

- *Sync_Hello*: it is used to notify a drone already in the network that a new drone has entered the system. It corresponds to the *computeCollision* request mentioned in Section 3.2. In particular, when a new drone joins the network, it spawns a drone_synchronizer process for each drone already in the system. A drone_synchronizer has the task to send *sync_hello* messages to another drone in order to share information about the route of the new drone, and so to know if there's a collision or not with the other drone.

  The structure of a *sync_hello* message is the following:

  ```
  {sync_hello, Pid, Id, MainPid, Route}
  ```

  In particular, a Route has this structure:

  ```
  {{Start_X,Start_Y},{End_X,End_y}}
  ```

  When a drone_synchronizer sends a *sync_hello* to a drone that is crashed, and so there won't be any response, after a timeout it retries to resend the message until the number of retries reaches a Retry limit (an environment variable of each drone that can be changed at compile time). When this limit is reached, the drone_synchronizer notifies the main process that there's no collision with the fallen drone.

- *Sync_Result*: It is used to reply to a *sync_hello* message, and it is directed to the drone_synchronizer process that is dealing with a specific synchronization for the main drone. It corresponds to the response *collisionDetected / collisionNotDetected* mentioned in Section 3.2. The goal of this message is to let the new drone know if there's a collision or not with another drone.

  The structure of this message is the following:

  ```
  {sync_result, External_Pid, External_Id,
      Collsion_response, Collision_point}
  ```

  where:

  - `External_Pid, External_Id` refer to the Pid and Id of the drone to which the new drone is synchronizing
  - `Collision_response` can contain the atom `collision` or `no_collision`

  - `Collision_points` is a map that, if `Collision_response` contains `collision`, contains the collision points both for the new drone and for the other drone that has computed the collision detection algorithm. This is needed since it's not always true that two drones in collision also have the same collision points (for instance, think the case in which a drone collides with the other only because one of its endpoints is too close to the other route)

- *Collision_Response*: Message sent from each drone_synchronizer to the main process when it receives the *sync_result* from the drone to which it was syncing to. Each drone_synchronizer terminates after this message is sent to the main process.

  The structure of this message is the following:

  ```
  {collision_response, External_Pid, External_Id,
        Collision_respone, Collision_points
  ```

  where

  - `External_Pid, External_Id` refer to the Pid and Id of the drone to which drone_synchronizer was syncing to.

- *Update_Table*: When the main process has received the *Collision_response* from each drone_synchronizer, this message is sent to all the drones with which a collision has been found. The effect of this message is to update the collision table of the receiver drone.

  The structure of this message is the following:

  ```
  {update_table, Pid, Id, Action, CollidingDrones,
                  State, Notify_count
  ```

  where:

  - `Pid` and `Id` refer to the sending drone of the message
  - `Action` can contain two different values:
    * `add`, that means that in the collision table the entry for the drone Id must be added or updated if already exists
    * `remove`, that means that from the moment in which the sending process has received the *sync_hello* message, to the moment in which the sending process has sent the *update_table* message, it has started to fly and its current position has changed making sure that there's no more collision with the receiver of the *update_table* message

At the end, the main outputs of this algorithm are two different data structure that are kept on each drone:

- **Collision Table**: is a map in which for each drone for which there's a collision, there's a map as value that contains:
  - `collisions`: A set of the id of all the drones that collide with him.
  - `notify_count`: a list containing all the drones to which it has sent a *notify* message.
  - `state`: is its state (`pending` or `flying`)

  The following is an example of how a collision table can appear:

```
#{
0 => #{
     collisions => sets:from_list([1, 2, 3]),
     notify_count => [],
     state => pending
},
1 => #{
     collisions => sets:from_list([0]),
     notify_count => [],
     state => pending
},
2 => #{
     collisions => sets:from_list([0, 3]),
     notify_count => [],
     state => pending
},
3 => #{
     collisions => sets:from_list([0, 2]),
     notify_count => [],
     state => pending
},
4 => #{
     collisions => sets:from_list([]),
     notify_count => [],
     state => pending
}
},
```

- **Personal Collisions**: is a map in which for each drone for which there's a collision, there's a map as value that contains:
  - `Pid`: The Pid of the drone.
  - `Points`: a map that contains the collision points with that drone.

  The following is an example of how Personal Collisions can appear:

```
            #{
            0 => #{
                 pid => <24062.468.0>
                 points => #{
                      0 => {23.0,23.0},
                      2 => {23.0,23.0},
                 }
            },
            1 => #{
                 pid => <24063.468.0>
                 points => #{
                      1 => {42.143,42.143},
                      2 => {42.143,42.143},
                 }
            },

        },
```

Priority Policy

The policy used by each drone to decide who is the next to have priority to fly is applied using a function (`compute_policy`) that is defined in a separate module (`drone_policy`). This separation allows users to easily change it, at compile time, in order to change the rules that drones follow when applying the policy.

Its definition is the following:

```
compute_policy(CollisionTable, Notify_Threshold)
```

where:

- `CollisionTable` is defined as shown in the previous section.
- `Notify_Threshold` is an integer that corresponds to the environment variable defined in the section about the drone hub.

It must return an ordering in the form of a list of the ids of the drones for which there's an entry in the collision table given as input. Our default definition follow the following rules:

- Drones with state "flying" always have the priority on all the others.
- If a drone D_1 has given the priority to other drones (i.e. it has sent a *notify* message) for a number of times greater or equal to the notify threshold, then D_1 will always have the priority on any other drones in collision with it (parities are broken by looking ids in ascending order)

- If two drones D_1, D_2, both in pending state and with a notify_count less than the threshold, are in collision but D_1 has a total number of collision less than the total number of collision of D_2, then D_1 has the priority with respect to D_2 (parities are broken by looking ids in ascending order)

Drone

Each drone process is structured as a finite state machine. In Figure 4.1 can be seen which are its main states and how a drone goes from one state to another. In the next sections, a brief description of each state is reported.

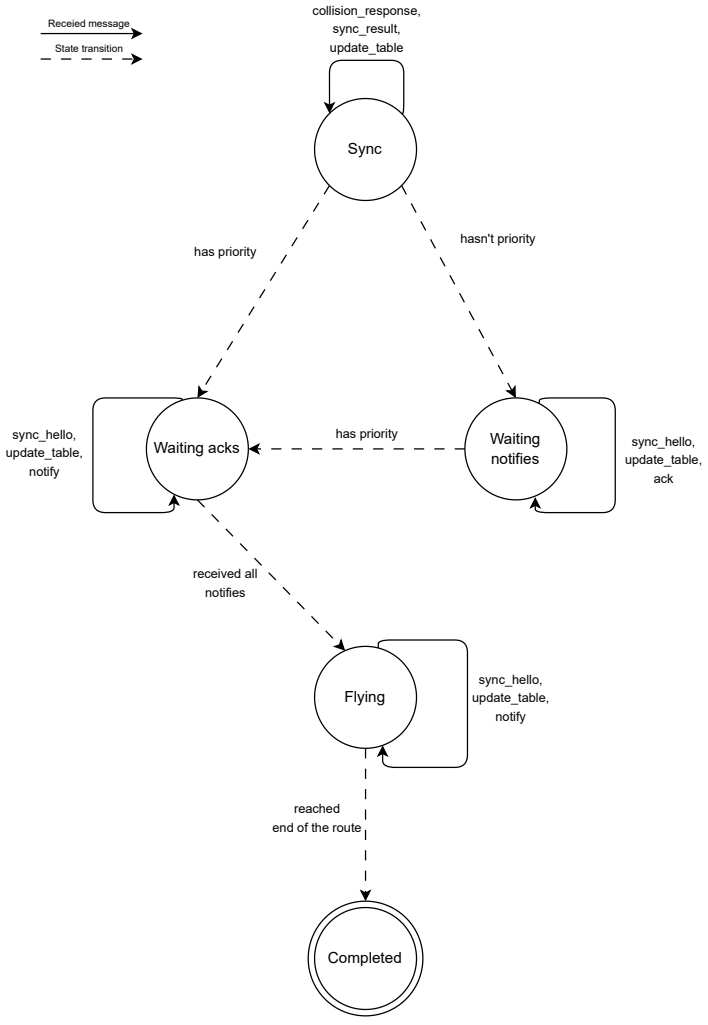

Figure 4.1: Finite state machine that represents how a drone process behaves

Sync state

This is the initial state for each drone when it enters the system. When a drone exits from this state it knows all its collisions with the other drones, and so, using the collision table built in this state, it can apply the policy and understand if it is the next drone to have priority or has to give priority to other drones.

In this state, initially each drone must call the Rest API asking for all the deliveries in the system that are not yet terminated. After receiving the list of the active drones (i.e. the drones dealing with deliveries yet not completed), it starts a drone_synchronizer for each of them. Moreover, each drone_synchronizer process is monitored from the main drone process in order to let the main process know if one of them fails.

In particular, if a drone_synchronizer process fails, the main process can simply respawn the corresponding drone_synchronizer. This is due to the fact that the last instruction of each drone_synchronizer is sending *collision_response* to the main process.

When all the *collision_response* messages, from each of the drone_synchronizer processes have been received, the main drone process knows all the collisions with the other drones already present in the system. Then, it sends an *update_table* message to each of the drones colliding with it and waits back the *update_table* message from them.

For each *update_table* message received in this state with Action `add`, a new entry is added in the collision table. When all the *update_table* messages have been received, the collision table that has been built can be used to apply the policy. Applying the policy to the collision table, an order of priority is given.

If the current drone it's at the head of the order, it means that it has the priority with respect to the other drones to which it collides and, so, it can pass to the **Waiting Notifies** state, otherwise it will check for the drones to which it can send a *notify* message and then goes to the **Waiting Acks** state. In particular, for each *notify* message that is sent, the corresponding drone's id is inserted in the notify_count (that is implemented with a list to check for duplicates every time a new id is inserted).

Waiting Acks

In this state, the drone waits for all the *ack* messages from the drones to which it has sent a *notify* message.

In particular, two different buffers are held by the drone: one to store the drones to which a *notify* has been sent in the current round (*Notified*), and one to store the drones from which an *ack* has been received in the current round (*ReceivedAcks*).

When all the acks have been received, the collision table is updated removing all the entries for the drones that have sent the *ack* (and also updating the collision field for the personal entry), then an *update_table* message is sent to all the drones that are still in collision with the current drone.

This *update_table* message will have the Action add, the state of the current drone, the current set of collisions and the updated *notify_count*.

After the propagation of the *update_table* message, the policy is reapplied to check who has now the priority. If the head of the resulting ordering is the current drone, it goes in the **Waiting Notifies** state, otherwise it remains in this state.

In this state, if a *sync_hello* message is received, the collision detection algorithm is used to detect if there's a collision and then, according to the result, a *sync_result* message is sent back.

However, it's possible that a drone receives a *sync_hello* message from a drone with an id that is already present in the collision table. In this case, it means that this message comes from the recovery drone that has been spawned from the supervisor due to a failure.

If this is the case, the drone must insert the id of the drone that has sent the *sync_hello* inside a buffer (*ToNotUpdate*) that contains the drones that are already known but have failed. This buffer is needed because, if the drone has received all the acks and then wants to propagate an *update_table* message to the drones to which it still collides, it must avoid sending it to a recovery drone that is still in its sync phase.

Moreover, when a failed drone is detected, if its id is included inside the *Notified* buffer, but not inside the *ReceivedAcks* buffer, the id of the failed drone is removed from the former.

When an *update_table* message is received, then there can be two different cases:

- The drone that has sent the *update_table* message is a new drone that is in the sync state. This means that the current drone has a new collision, so it adds a new entry in its collision table, and sends an *update_table* message to each drone already in collision with him (but not to the drones from which is waiting an *ack* message). Moreover, if the number of *notify* messages that have been sent from it is less than the notify threshold, it will also check if it can send a *notify* message to the new drone.

- The drone that has sent the *update_table* message is an already known drone. However, it is possible that this message comes from a recovery drone. To handle this case, it is checked if its id is included in the *ToNotUpdate* buffer. If it is included, first an *update_table* message is sent back and then the drone's id is removed from the buffer. Then, in any case, in the collision table must be updated the corresponding entry and, if the number of *notify* messages that have been sent from it is less than the notify threshold, it can be checked if a *notify* message can be sent to it.

Waiting Notifies

In this state a drone must wait for the *notify* message from all the drones with which it collides. Obviously, if it isn't in collision with any other drone, it can

directly pass to the **Flying** state.

Like in the **Waiting Ack** state, also in this state a *sync_hello* can always be received from a new drone that enters the system, and it is handled using the collision detection algorithm in order to detect if there's a collision and then returning back a *sync_result* depending to the output of the algorithm.

However, if a *sync_hello* is received from someone from which the drone is waiting a *notify*, this means that the *sync_hello* comes from a recovery drone, so it is possible that this drone before the failure had still sent the *notify* message. In this case, its id is also removed from the *ToBeAcked* buffer, in which are stored the id of the drones from which a *notify* message has been received.

If a drone, in this state, receives an *update_table* message, there can be two different cases:

- The *update_table* message comes from a new drone that is in the sync state, so the current drones won't have an entry for it in its collision table. However, since the current drone has already taken the priority, it's possible that if the new drone computes the policy, then the priority moves from the current drone to the new drone.

  In order to avoid this a drone, when in this state, buffers the *update_table* message received from each new drone and applies them to its collision table only when it has received all the needed *notify* messages and starts to fly. Moreover, when it applies each of these buffered *update_table* messages, it will also send back an *update_table* message to each of the drones that have sent the buffered *update_table* message.

  Doing this, allows that when the new drone has received all the *update_table* messages and has applied the policy to its collision table, it surely will notify the current process since its state is flying.

- The *update_table* message comes from a drone for which there's already an entry in the collision table. This can mean two different situations:

  - It comes from a drone from which the current drone is waiting for a *notify* message or has already received a *notify* message, it has been sent because the number of collisions of the sender drone has increased. Since the current drone will always have fewer collisions than the sender, it can stay in the current state.
  - It comes from a failed drone, from which the current drone has already received the *sync_hello*, and moreover it already has an entry for it in its collision table.

  In both these two cases, the drone can send back an *update_table* message, since for the first case, it cannot be a problem (at most a duplicated *notify* is received), while in the second case it will simply allow the failed drone to complete its synchronization.

When all the *notify* messages have been received, the drone can take off and start to fly, passing to the **Flying** state.

Flying

When arriving in this state, a drone, first of all, must spawn a **Flight** process.
This latter process has the tasks to:

- Update the current drone position, sending to the main process the new current position each time it is computed.
- Send a message to the drone process when the collision point with one of its colliding processes has been passed, in order to allow the drone process to send an *ack* message to it.

In particular, when the Flight process is spawned, it will be also monitored from the main drone process in order to know if it crashes and, spawning a new process in this case.

As in the Waiting Ack and Waiting notifies state, when a *sync_hello* message is received, it will be handled in the same way returning a *sync_result*.

When a drone in Flying state receives an *update_table* message, two different cases must be distinguished:

- The *update_table* message comes from a new drone that is in the sync state. In this case, the drone must check if the collision point with the new drone has already been passed. If this is the case, the current drone sends back an *update_table* message but with Action remove. In the other case, it sends back a normal *update_table* message with Action `add`.
- The *update_table* message comes from a known drone. In this case, the only possible case is that it comes from a recovery drone. So, the current drone will check if the collision is passed, and depending on this will send an *update_table* message with Action dd or remove.

In this state it is possible to receive *notify* messages from drones that synchronized with the current drone while it was already flying.

When a *notify* message is received, there are two different cases:

- The collision point with the drone that has sent the *notify* message has been already passed. So, in this case, the current drone immediately sends back the *ack* message to this drone.
- The collision point with the drone that has sent the *notify* message hasn't been yet passed. In this case, this drone is added to a buffer containing all the drones for which an *ack* message must be sent, and, in order to send back the *ack* message when needed, every time the Flight process update the current position of the drone, it must be checked if has been passed any of the collision points for the drones contained in this buffer.

Chapter 5

Validation

In this section we will talk about both unit testing that we did and the validation. In particular, for the testing we will present some tests done using the framework of unit testing Eunit, and then for the validation, using some simulations, we will show that the system fulfills the requirements analyzed in chapter 2.

5.1 Unit testing

We mainly used unit testing to test some crucial aspects that each drone must satisfy: computation of a potential collision with another drone, computation of the policy in order to decide if a drone must give priority to others or has priority in the current round, and building of the collision table for each drone at the end of the synchronization phase with all the others.

Collision computation

Using Eunit we were able to write some significant cases for collisions detection between drones. In particular, we showed that our algorithm for collision computation is able to detect when there's a collision between two parallel routes since there can be a potential collision between the body of the two drones in the case that the drones are flying at the same time.

Moreover, we tested that the algorithm is able to find a collision when two routes are neither parallel or in an intersection but one of the endpoints of a route is too close to the other route, to the point that the bodies of the two drones can collide if flying at the same time.

Further, we showed that the algorithm works well also in the trivial case in which there's an intersection between the routes of two different drones.

Our test cases also checked that the collision points returned by the algorithm are exactly the ones that we expect.

Policy application

For this aspect, we tested which was the output of the policy function used by each drone according to the collision table passed as input.

We checked the ordering given as output for the case in which all the drones that are in collisions to the drone that is computing the policy have an equal number of collisions and the discriminant to decide the priority must be the id of each drone.

We also checked the case in which there's a drone for which the *notify_count* in its state exceeds the fixed threshold and so, if it's the only drone for which this guard is true, then it has the priority to fly.

Moreover, we tested the case in which there's already a flying drone with which there's a collision and so, the already flying drone has always the priority.

It's important to say that since the policy used by each drone can be changed simply by editing the *compute_policy* function exported by the *drone_policy* module, if this is done our defined test cases cannot succeed since they are written according to the default policy that we have defined.

Synchronization phase

Since this aspect concerns the communication between drones, to test it we had to change a bit the workflow of our algorithm since the collision table that each drone builds for himself is part of its state and cannot be accessed by external processes unless it is sent in a message.

So, to solve this problem we took the module that concerns with the synchronization (*drone_main*) and wrote a test version of it in such a way that whenever a drone terminates the building of its collision table, or receive an *update_table* message from a new drone, it sends the table to a testing process. So, after the testing process collects all the received tables, then it checks if their content is equal to the one expected.

Using this workaround, we checked four significant cases: one in which five drones are not in collision with any other, one in which there are four drones and for each of them there's always at least a collision with another, and one in which there's a ring of collisions between four drones (i.e. drone 0 collides with 1 and 3, 1 collides with 0 and 2, 2 collides with 1 and 3, 3 collide with 2 and 0).

5.2 Requirements checking

Using the UI control service that we implemented, we are able to show that the requirements that we first presented in Chapter 1 and then analyzed has been fulfilled. In the next sections, showing some simulations of the systems, we will demonstrate the fulfillment of the following requirements:

- **Fairness**: the drones behave in a fair way. The policy computation done by each drone allows to have each time a fair scheduling of who are the next drones that have priority to fly.
- **Starvation**: The system behaves in such a way that isn't possible to have a drone that waits forever in order to start to fly.
- **Fault tolerance**: Even if a drone falls, the system guarantees that a new drone is spawned from the last position reached from the failing drone in order to complete the delivery.
- **Deadlock**: The agreement protocol guarantees that it's not possible that two or more drones wait for each other in order that someone of them can start to fly.

Fairness

To show that this requirement is satisfied, we did a situation in which there are five drones, where four of them collide with at least another drone while the last drone isn't in collision with any other and can start to fly immediately. This scenario is illustrated in Figure 5.1.

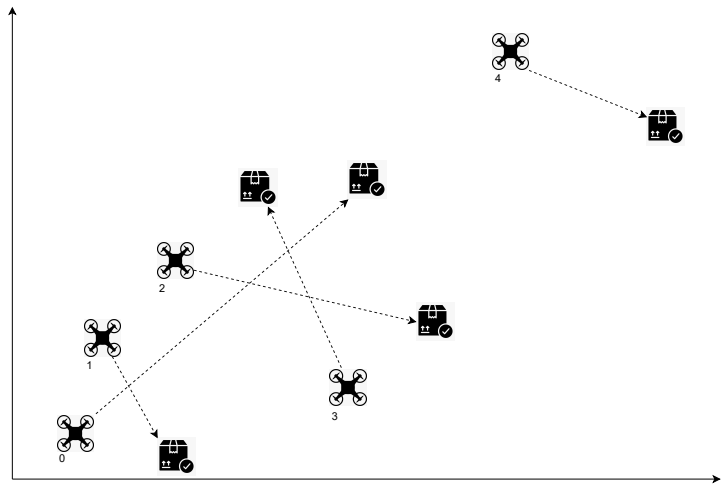

Figure 5.1: Location of drones in scenario used to check fairness requirement

Since deliveries are inserted from the UI control service one at a time, the drone hub spawns corresponding drones in the order in which it receives the request of insertion. This means that when the first delivery (0) is inserted, the drone that handles it is the only one in the system and so it isn't in collision with any other drone. So, drone 0 can start to fly immediately.

Then when 1, 2 and 3 enter the system, since they are all in collision with 0, they must send notify to it and wait for the respective acks. Since 1 will be the first that receives the ack from 0, then it will also start to fly.

At the same time, when drone 4 joins the network, since after the synchronization phase it sees that it isn't in collision with any other drone, it can start to fly immediately.

When drone 0 sends the acks to drone 2 and 3, then 2 is the first to start to fly since in its collision table the only one remaining drone is 3 that has a greeter id.

The last drone that starts to fly will be the drone 3. In Figure 5.2 can be seen the simulation of this scenario.

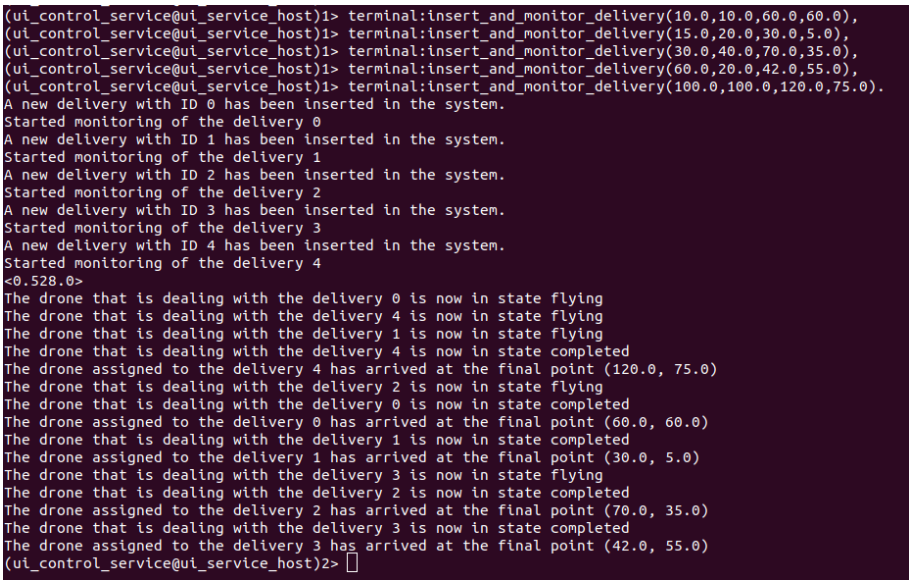

Figure 5.2: Simulation of the behaviour of drones to check fairness requirement

Starvation

This requirement has been checked using a scenario with seven drones in which six of them collide only with one common drone, that is the only one that collides with all the others. This scenario is illustrated in Figure 5.3.

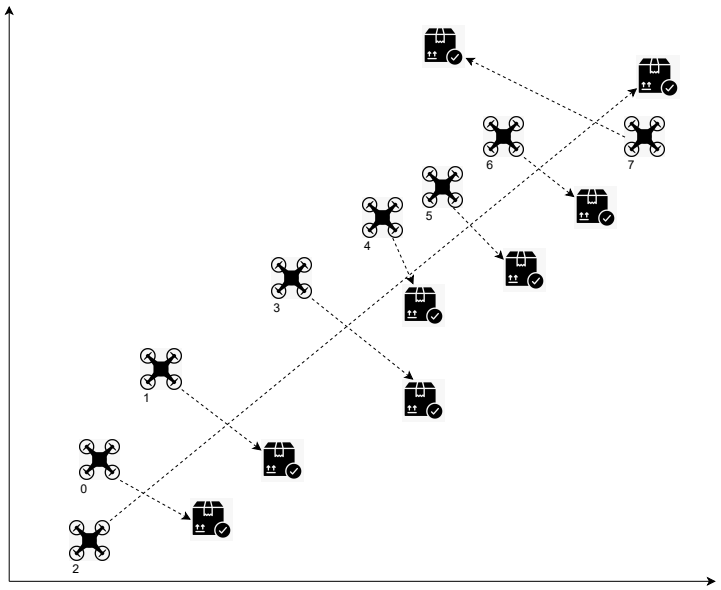

Figure 5.3: Location of drones in scenario used to check starvation requirement

At the start, when drones 0 and 1 are inserted, since there's no collision between them, they can start to fly immediately.

When drone 2 joins the network, it cannot start to fly since it is in collision with both 0 and 1 that are already flying. So, 2 must send notify to 0 and 1 and wait for the respective acks. Moreover, when drone 3 and 4 join the network, since they collide only with 2 that is already waiting acks from 0 and 1, they have priority with respect to 0 since they have only one collision, so 2 sends notify also to 3 and 4.

In this way, 0 has now a *notify_count* equal to 4 (that in this case is also the *Notify_Threshold*) and cannot send any other notify to other drones.

So, when 5, 6 and 7 join the network, since from the synchronization phase they will know the state of drone 2, they see that 2 has reached the notify threshold. This means that for them now 2 has the priority.

In this way, when 2 receive the acks from 0, 1, 3 and 4 and then receive notify from 5, 6 and 7, it can start to fly. At the end, drones 5, 6 and 7 will start to fly only when they receive the ack from drone 2. The simulation of this scenario can be seen in Figure 5.4.

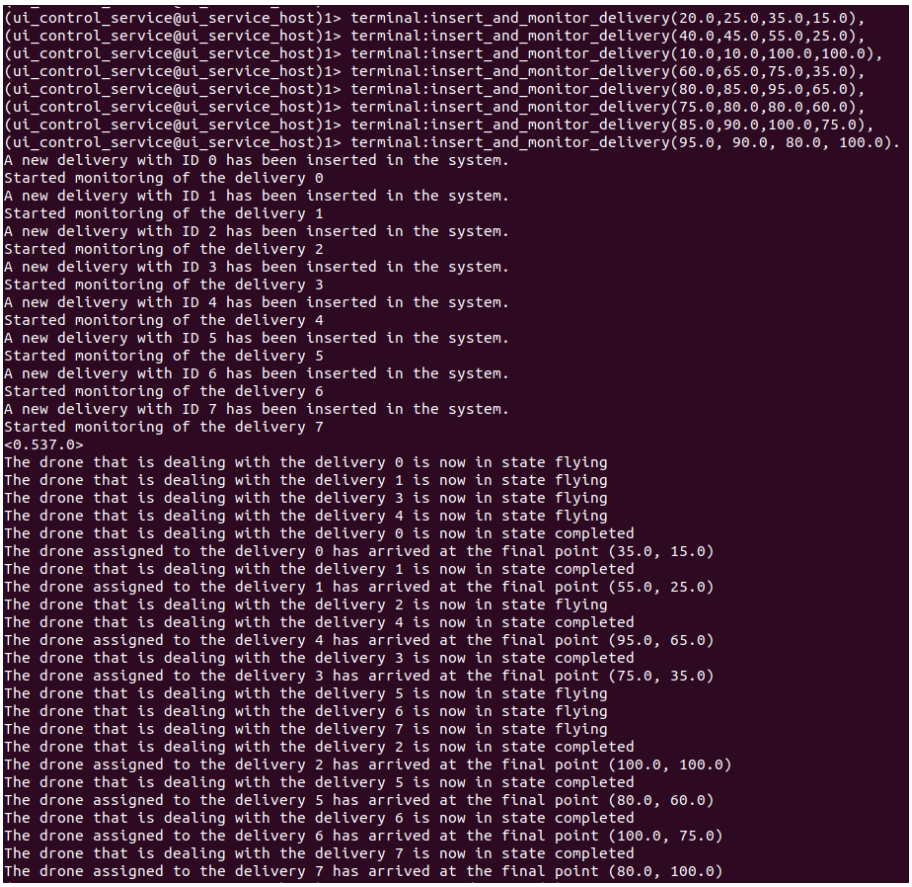

Figure 5.4: Simulation of the behaviour of drones to check starvation requirement

Fault tolerance

To validate this requirement it can be used the same scenario depicted in Figure 5.1, but killing a drone.

For instance, the kill of drone 4 won't be really significant since it isn't in collision with any other, while killing drone 0 is much more interesting since its failure should also affect drone 1, 2 and 3.

In Figure 5.5 a simulation of this scenario can be seen.

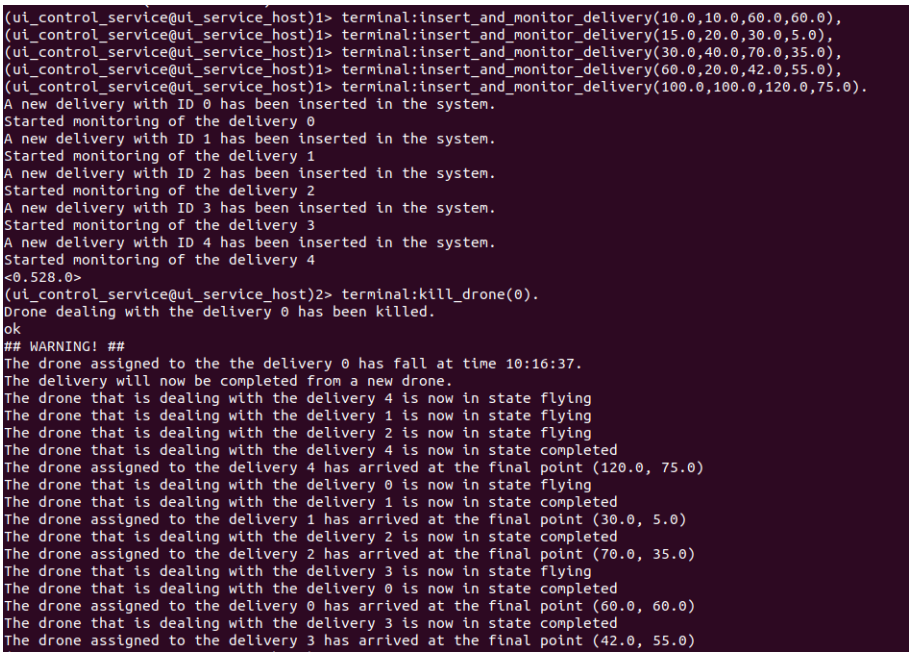

Figure 5.5: Simulation of the behaviour of drones to check fault tolerance requirement

This time, as opposed to the simulation done for the fairness validation, after the failure, drone 0 isn't anymore the first drone to fly with respect to 1, 2 and 3, but actually when 1 and 2 understand that drone 0 has failed they start to fly.

Then, only when it receives the ack message from 1 and 2, drone 0 can also start to fly while drone 3 gives priority to drone 0 since its id is greater than 0. As mentioned before, anyway the fail of 0 doesn't affect drone 4 since it isn't in collision with anyone.

Deadlock

Validation of this requirement has been done using the scenario represented in Figure 5.6, where drone 0 collides with 1 and 3, drone 1 collides with 0 and 2 and so on.

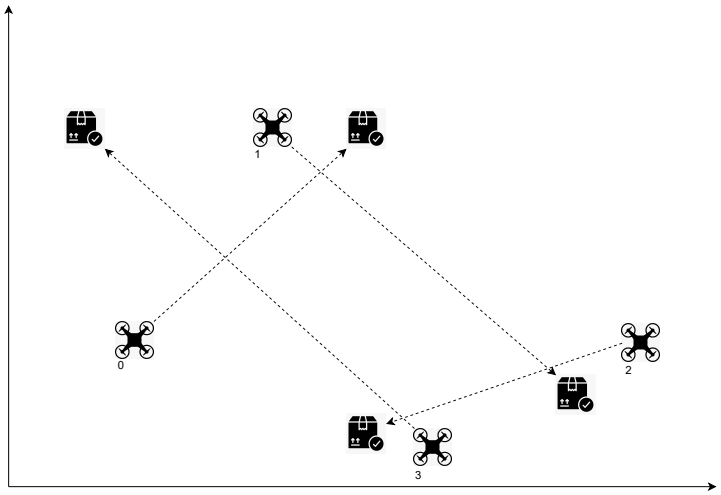

Figure 5.6: Location of drones in scenario used to check deadlock requirement

As already said for the validation of fairness, since each drone is spawned one at a time in the system, when drone 0 joins the network actually it is alone, so it can start to fly.

Then, when drone 1 joins the network, 1 must wait until 0 sends back the ack to it. However, if drone 2 joins the network before that drone 1 receives the ack from 0, then 1 can also notify 2 and 2 can start to fly. When 3 joins the network, since it collides also with 2 and 2 is already flying, it must wait the ack from 2 and also from 0 in the case that 3 was still in collision with 0 when joined the network.

So, in this scenario we have that 0 and 2 are the firsts to start to fly, and then when they send back the aks, 1 and 3 also start their flight.

In Figure 5.7 can be seen the simulation of this scenario.

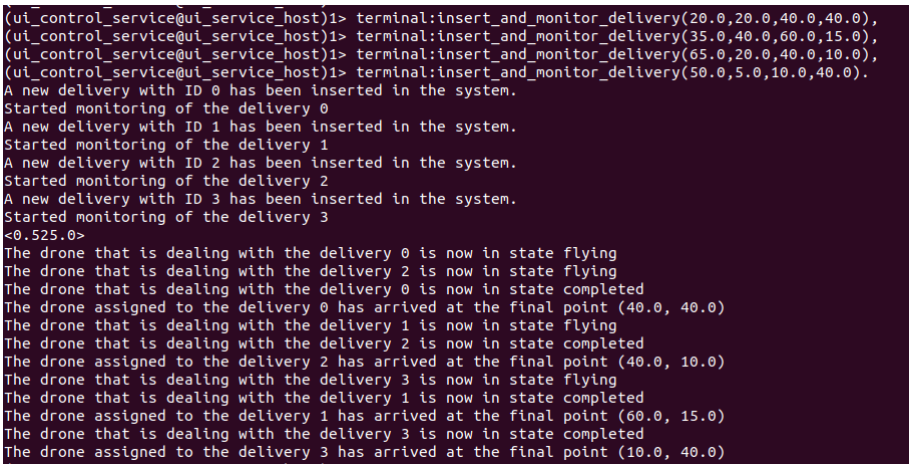

Figure 5.7: Simulation of the behaviour of drones to check deadlock requirement

Chapter 6

Conclusions

Our solution to the problem implements all the functional and non functional requirements analyzed in Chapter 2. Considering the tiers that we have defined in the development plan, our solution fulfills what we have defined in tier 4.

However, some future developments can be thought. For instance, the Rest API that we have defined could be used by a GUI to allow a simpler insertion of new delivery into the system or also to implement a graphical visualization of the drones in the ground.

Another possible future development can be to add a third dimension for the space in which drones fly.

Appendix A

A.1 Instructions

To use the distributed system, first of all, the docker image of the drone must be built.

Locate to `./src/drone` directory and then executes the following command:

```
./docker-build
```

Then to start all the required components of the distributed system, locate to `./src` directory and then executes the following command:

```
docker-compose up
```

`--build` option can be added to the command above, to rebuild the images of all the components.

To interact with the system, using the UI Control Service, locate to the directory `./src/ui_control_service` and run the following commands:

```
./docker-build
./docker-run
```

A.2 Configuration

The configuration of the system can be changed editing some environment variables:

- `DEV_MODE`: when is enabled the drones are spawned in the same node of the drone_hub otherwise every drone will have its own container.
- `VELOCITY`: speed of flight for the drones (m/s).
- `DRONE_SIZE`: Size of the bounding box.
- `FLY_HEIGHT`: height to which drones fly.
- `NOTIFY_THRESHOLD`: Threshold that allow a drone to get priority against the other drones.

- `RETRY_LIMIT`: Max number of retries that a drone do in order to synchronize with other drones(also defined in the Dockerfile of drone).
- `HOST_PATH_VOLUME`: directory to which logs files are stored.

All the variables mentioned above can be found in the `drone-hub-variables.env` file located inside the `./src` directory.

If the `RETRY_LIMIT` variable is edited only on `drone-hub-variables.env`, the change will have effect only when the `DEV_MODE` is enabled. Otherwise, when `DEV_MODE` isn't enabled, the change must be done inside the Dockefile in `./src/drone` and then the image must be rebuilt.

Regarding the Rest API, the following variable can be change:

- `MAX_SIZE`: size of the Cartesian plane

The variable above can be found in the `rest-variables.env` file located inside the `./src` directory.

In order to change the policy that drones use to make the priority agreement, the function `compute_policy(CollisionTable, Notify_Threshold)` located in `./src/drone_hub/src/drone_policy.erl` must be changed.

This change will have effect only when the `DEV_MODE` is enabled. Otherwise, the same function defined in `./src/drone/src/drone_policy.erl` must be changed, and the image of the drone must be rebuilt.

A.3 UI samples

Inside the directory `./src/ui_examples` can be found some files containing commands that can be inserted on the UI Control service to simulate the scenarios that have been used in Chapter 5.