SAPIENZA
UNIVERSITÀ DI ROMA

DEPARTMENT OF COMPUTER SCIENCE

IoT Project: version C

INTERNET OF THINGS

Professors:
Novella Bartolini,
Federico Trombetti

Students:
Cesare Corsi,
Andrea Massignan,
Kevin Cukaj

Academic Year 2023/2024

Contents

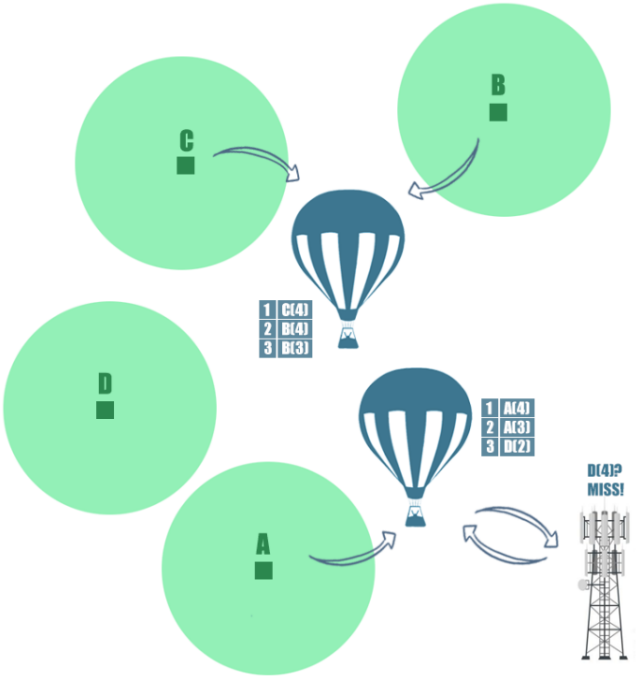
1 Introduction

1.1 Overview of our personal approach

The version C of the project requires **static balloons** and **mobile sensors**. Starting the simulation for the first time we noted that the default scenario presented us with just a single balloon and three sensors. The project specifications required us to test in different scenarios on how to best achieve a behaviour that could represent the core of the project's request.

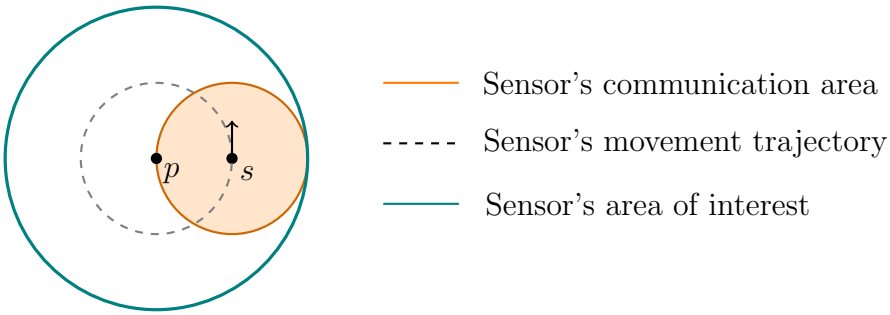
We initially thought of different ways to implement the algorithm for determining the spawning positions of the balloons given n sensors. The sensors' position was given by a pseudo random (according to a specific probability distribution) algorithm making them spawn with enough distance from each other and the **base station**.

Once we reached an agreement, we shifted our attention on the mechanics of how sensors should move by pairing them with an **Action Client**. Since the balloons should embed a cache-replacement policy we then decided on how to implement it. Lastly we configured the logic behind the base station requests, using the approach of an **Action Client/Server**, and how to evaluate the performance of the system.

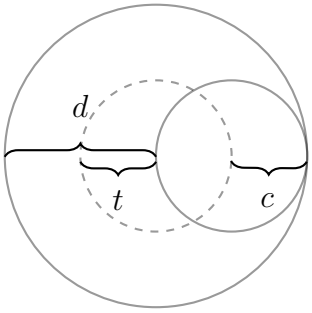


1.2 Sensor's motion

In this section we discuss the movement of the sensors. They have an associated communication range c, and we make them move in a circular trajectory around a corresponding point p.



Each of the three formed circles has its respective radius. We will denote them with d, t, s. Meaning,



Clearly, $d = t + c$. In our simulations, we impose that $t \leq c$. This way, if we place the balloon anywhere inside the blue circle with radius d, there will be a time window where the balloon will be able to receive data from the sensor by being inside its communication range.

2 Development

2.1 Algorithm for the positioning of the balloons

In the following, we discuss the main idea behind the algorithm for determining the placement positions of the balloons given the positions of a set of sensors. Each sensor is characterized by its x and y coordinate values.

The algorithm starts by considering all possible pairs of sensors. For n sensors, there are $\binom{n}{2}$ such pairs. For each pair (s_i, s_j), the algorithm considers the distance between the two given by $d = \sqrt{(x_i - x_j)^2 - (y_i - y_j)^2}$. If they are distant below a given threshold, then the algorithm notes down that the *areas of interest* of the two sensors intersect. By the end of this phase, we have a set S of the kind

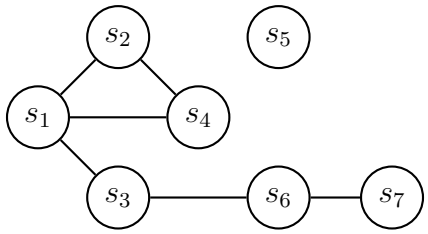
$$S = \{(s_1, s_2), (s_7, s_3), (s_5, s_9), ...\}.$$

In other words, this set denotes that sensor s_1 is *very close* to sensor s_2, sensor s_7 to sensor s_3, and so on. This is an important information because it tells us that we can place a balloon in the middle point between two close sensors and listen to them at the same time.

In the next phase, the algorithm build an undirected graph G given set S. The graph will have a node for each sensor. Two nodes s_i, s_j will have an edge between them if and only if $(s_i, s_j) \in S$. For example, for

$$S = \{(s_2, s_4), (s_1, s_3), (s_1, s_2), (s_1, s_4), (s_3, s_6), (s_5)\},$$

the resultin graph will be the following.

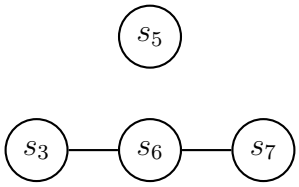


After building the characteristic graph G from S, the algorithm consider all possible length-3 cycles. For each of such cycles, there will be a single balloon being able to listen to the three sensors at the same time.

To find such cycles, the algorithm considers all possible triples of elements in S, say $\sigma_1 = (i, j), \sigma_2 = (i, k), \sigma_3 = (j, k)$. If $|\{\sigma_1 \cup \sigma_2 \cup \sigma_3\}| = 3$, then G will have a length-3 cycle made up from the nodes s_i, s_j, s_k.

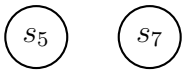
Consider again the graph illustrated above. When the algorithm finds the length-3 cycle made from the nodes $\{s_1, s_2, s_4\}$, those nodes along with the edges touching

any of them, get removed from the graph. Meaning, the resulting graph will be the follwing.



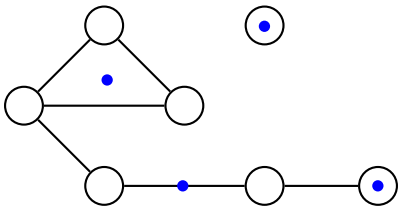
After removing all length-3 cycles, the algorithm enters the second phase, in which it looks for couples. When it finds one, it removes it along with all the edges like in the previous phase. For each couple, there will be a single balloon.

If we continue to consider the previous graph, the algorithm will find a couple; say (s_3, s_6). The resulting graph will be the following.

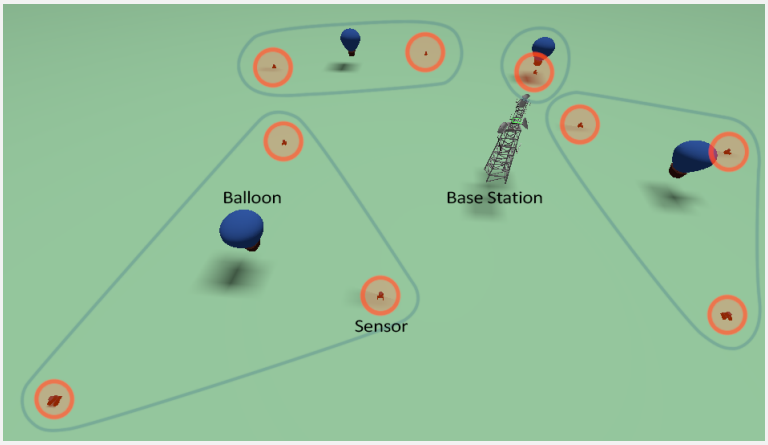


In the third phase, the algorithm considers the remainging isolated nodes, which represent sensors that are too far from all the others. For each of them, there will be a balloon.

To determine the exact position of the balloons in the plane, their x and y coordinates are obtained by calculating the medium points. Here's where the balloons would be positioned for the graph we've been considering so far.

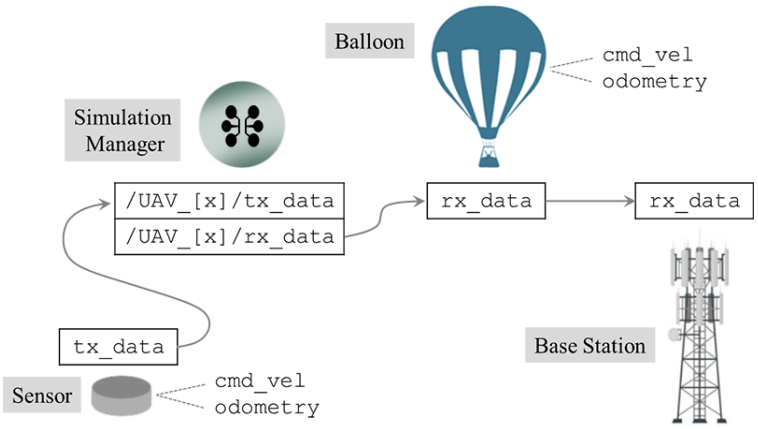


And here's what it would look like in Gazebo. (Note that it is not the same graph.)



2.2 Topics overview

In the following image we can see a general overview of the simulated elements with the topics used to manage them, and to let them communicate with each other.



2.3 Sensor movement implementation

In order to make the sensors move, we created an Action Client (`sensor_action_client`). The Node receives the number of sensors from the `simulation_launch` file and create a single Action Client that manages each sensor.

Initially, the Node receives the number of sensors along with their initial positions and stores this information in a list, as does each Action Client. During this process, the initial positions of the sensors are passed to the dedicated function `math_utils.construct_goal_points`, which generates a list of lists of points. Each sublist corresponds to the path that the associated sensor will follow. The points are constructed by generating a circle of radius r centered at the initial position of each sensor.

After the initialization, the Node starts by executing a timer that calls onto the `self.goals_duty_cicle` function. Since we are also storing all the futures of each goal within `self.sensor_result_futures` that is constructed by each request of the Action Client, we can check if the goal has been completed or not. If it has, we can send the next iteration of goals to the sensors. At the beginning the list will be surely empty, so the first iteration of goals starts.

When the Action Server on `sensor_controller` receives a goal, it starts the execution of the movement of the sensor. The patrolling is done by moving the sensor from one point to the next in the list of points. It adapts a policy of first rotating the sensor towards the next point and then moving towards it. The sensor will keep moving until it reaches the last point in the list. When it does, it will return a success message to the Action Client while terminating its movement for the time being.

When the Node receives the completion message from all the sensors, it will send the next iteration of goals to them. This process will continue until the simulation is terminated.

2.4 Balloon cache replacement policy

The sensors produce data that has the following format: `sensor_id`, `data`, `time_found`, `expiration_time`.

The cache replacement policy is implemented in the `balloon_controller` Node. Differently from what it was done for the sensors, we have created multiple Nodes that handles the behaviour of the balloons. Each Node is responsible for a single one of them.

The Node receive the `id` of the balloon from the `simulation_launch` file, it creates its own Action Server to receive requests from the base station. It has the subscription to both its own `odometry` topic and the `rx_data` topic to receive the data from the sensors (handled by the `simulation_manager` node).

The only movement action that the balloon can perform is to rise up in the z direction at a default altitude of 1 unit. This is done by the `fly_to_altitude` function that is called when the balloon is first initialized.

The cache replacement policy follows the `Least Recently Used`. For implementing it, we created a class named `LRUCache` that is responsible for storing the data received from the sensors. We agreed to set the length of the cache to the reasonable length of 5 slots to simulate a memory constraint. When the cache is full and a new data is received, the oldest used one is removed. When the balloon receives data from the sensors by the `rx_data` topic, it stores the data with the associated `sensor_id` in the cache with the `rx_callback` function.

2.5 Base station request handling

The base station is responsible for sending requests to the balloons. The Node receives the number of sensors and the number of balloons from the `simulation_launch` file and then creates an Action Client for each balloon.

The base station is equipped with a number of variables that mantains the correct routine of its requests:

- A variable that informs the base if it is currently waiting for a response (`self.is_asking`).
- A variable to keep track from which sensor is currently expecting new data (`self.interrogating_sensor`).
- A database in which the data received from the balloons is stored (`self.database`).

- A dictionary of timestamps to later evaluate the delay of the system (`self.timestamp_dict`).
- A list to store the different Action Clients for each balloon (`self.balloon_clients`).
- A list to store the future of each request sent to the balloons (`self.balloon_result_futures`).

When the base station is correctly initialized, it starts a timer that calls the `self.managing_data` function every 5 seconds. The function checks if the base station is currently waiting for a response from the balloons. If it is not, it handles the logic of sending requests to the balloons by considering the following aspects:

- It checks the length of the database. If it is equal to the number of sensors, it means that the base station has received data from all the others and it can proceed to send the requests to the balloons by selecting the sensor that has the lowest value inside the database. Otherwise it randomly selects a sensor from the `ids` list.
- The `random_ask_for_data` or `get_specific_data` is called accordingly.
- The base station sends the request to all the balloons, as it doesn't know the position of any of them.

When the goals are sent, it waits for the response from the balloons. Evaluation of the delay of the system is done by calculating the difference between the time that the base station sent the request and the time it received the response. This is done considering the timestamps stored in the `timestamp_dict`. Two types of evaluation are done:

- The delay of the packets moving from the balloons to the base station.
- The validity of the data received based on the expiration policy, if the data is received after the expiration threshold, the data is considered outdated.

If the balloons respond with a `result` containing the data, and it is not outdated, the base station stores the latter in the database. Otherwise if the balloons responds with a `Data not found` message, for a fixed number of times (3), only if the database is complete with all the sensors `ids`, the base station keeps asking for the same sensor's data, otherwise it will ask to retreive data from another sensor randomly.

3 Performance evaluation

For this particular scenario we came to the conclusion that the only useful metrics for measuring the performance of our solution would be the following:

- **Data Loss**: When the base station requests data from a sensor, the balloon should be able to retrieve it and send it back. However, if the data is not found inside its cache, then it will count as a data loss.
- **Packet Delay**: The time between the request of the data from a balloon and the reception from the base station.

Due to the high computational cost of the simulation, we decided that the simulation would be run for an interval of 10-15 minutes.

The following tables represents the average performance for multiple simulation instances based on the number of sensors:

# of Balloons	**Cache size**	**Mean Time Delay (s)**	**Data Loss (%)**
3	5	0.0102	6.1224
3	5	0.0443	3.3898
4	5	0.0116	5.6338
3	10	0.0102	1.2346

Table 1: 5 Sensor Scenario

# of Balloons	**Cache size**	**Mean Time Delay (s)**	**Data Loss (%)**
5	5	0.0550	5.6603
5	5	0.0250	4.0540
6	5	0.0926	6.4102
4	10	0.0864	7.0175

Table 2: 10 Sensor Scenario

# of Balloons	Cache size	Mean Time Delay (s)	Data Loss (%)
7	5	0.1093	20
8	5	0.0304	15.5172
7	5	0.0197	25.9259

Table 3: 15 Sensor Scenario

More specifically we can observe the trend of the `mean time delay`, for the different scenarios in the following graphs:

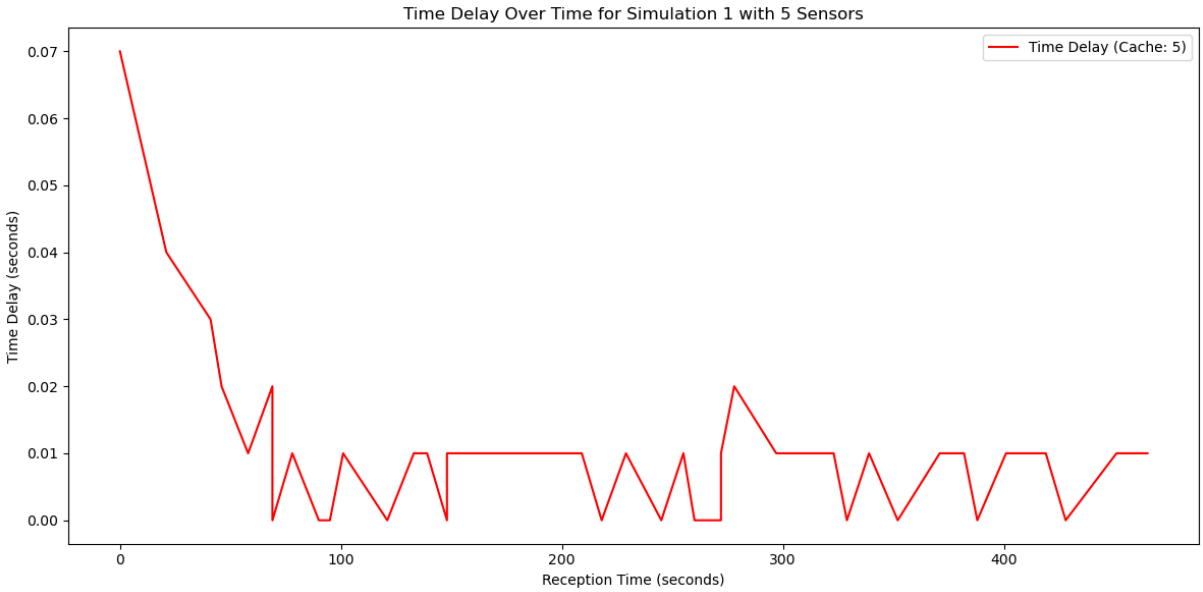


Figure 1: Simulation 1 - 5 Sensors

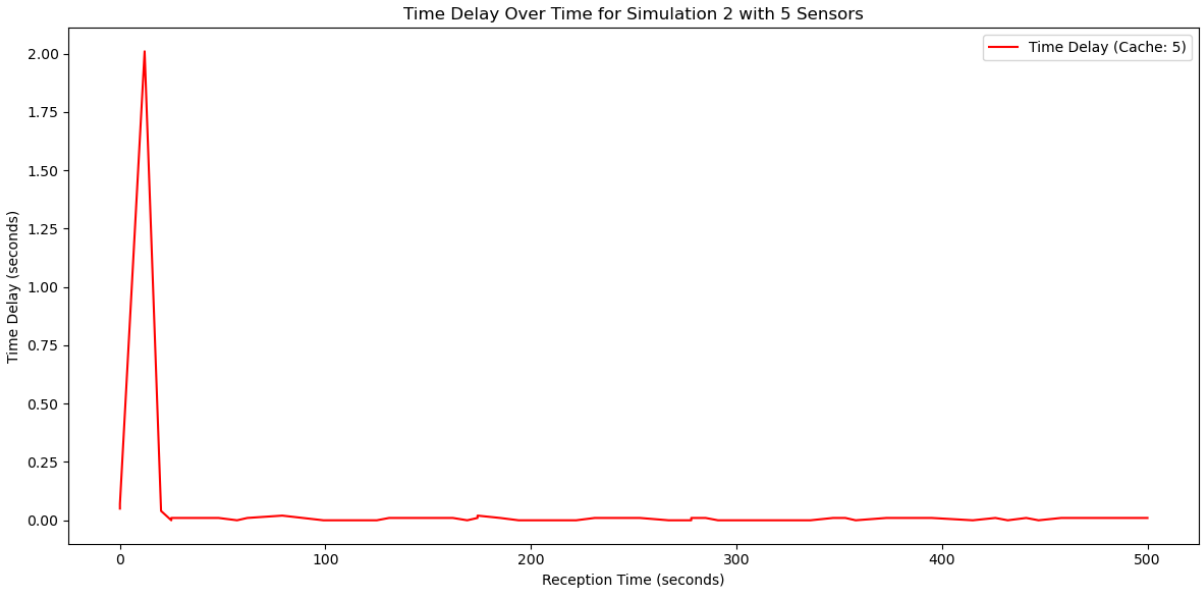


Figure 2: Simulation 2 - 5 Sensors

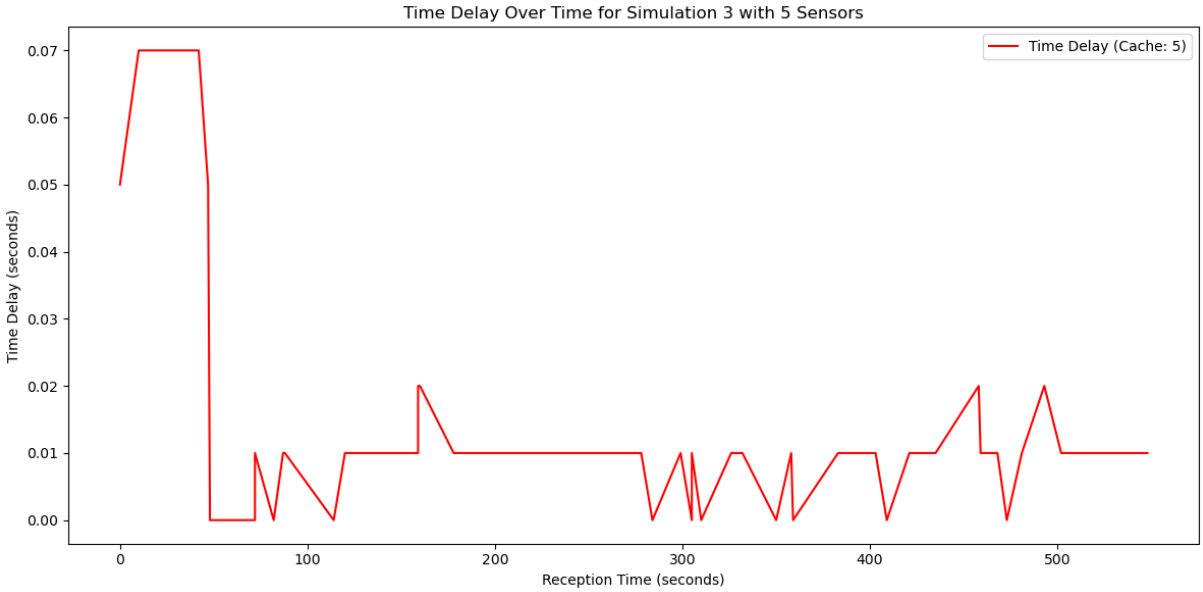


Figure 3: Simulation 3 - 5 Sensors

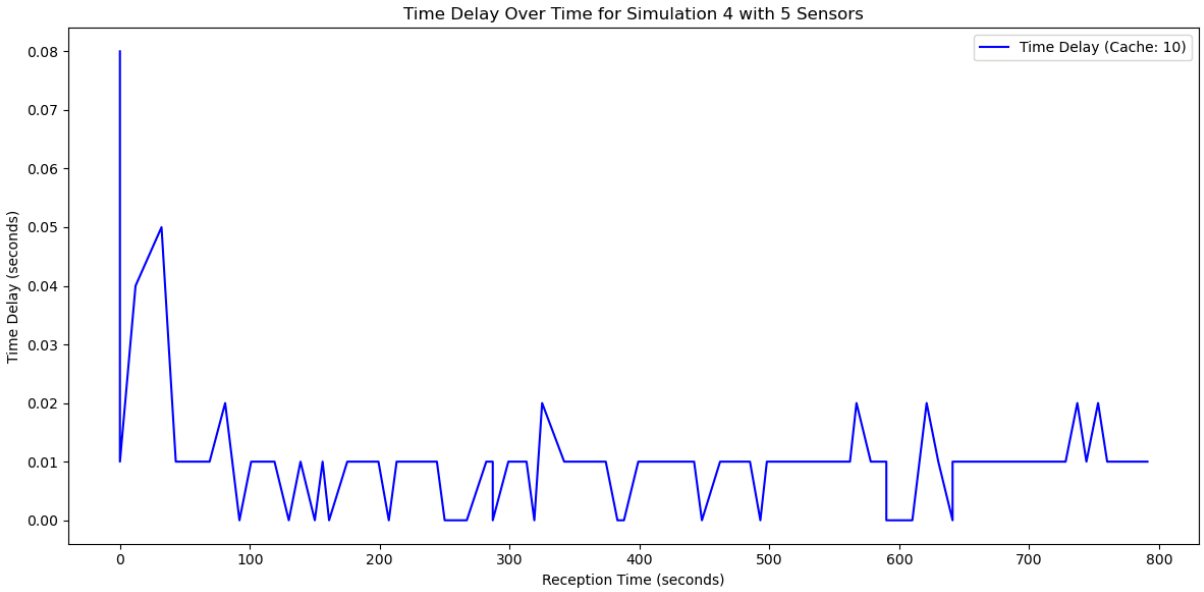


Figure 4: Simulation 4 - 5 Sensors

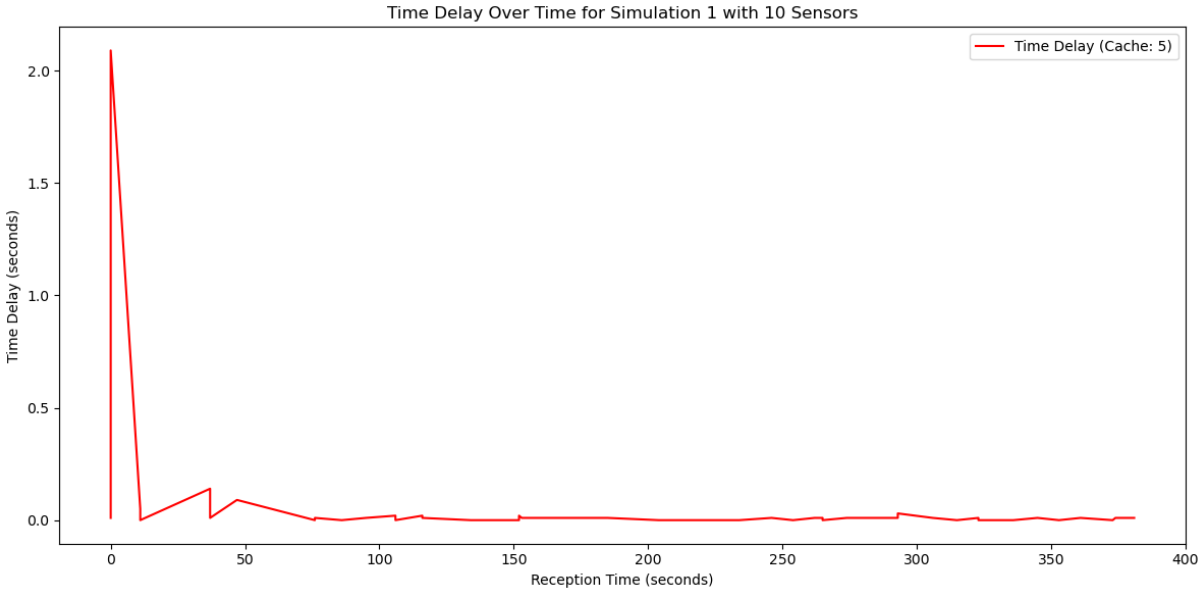


Figure 5: Simulation 1 - 10 Sensors

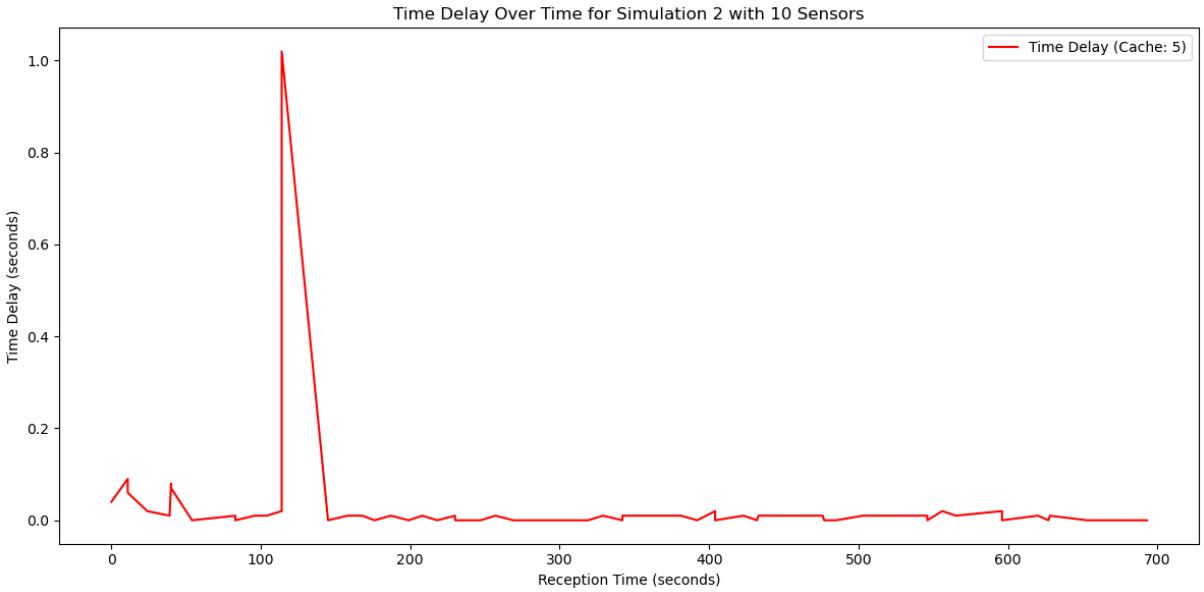


Figure 6: Simulation 2 - 10 Sensors

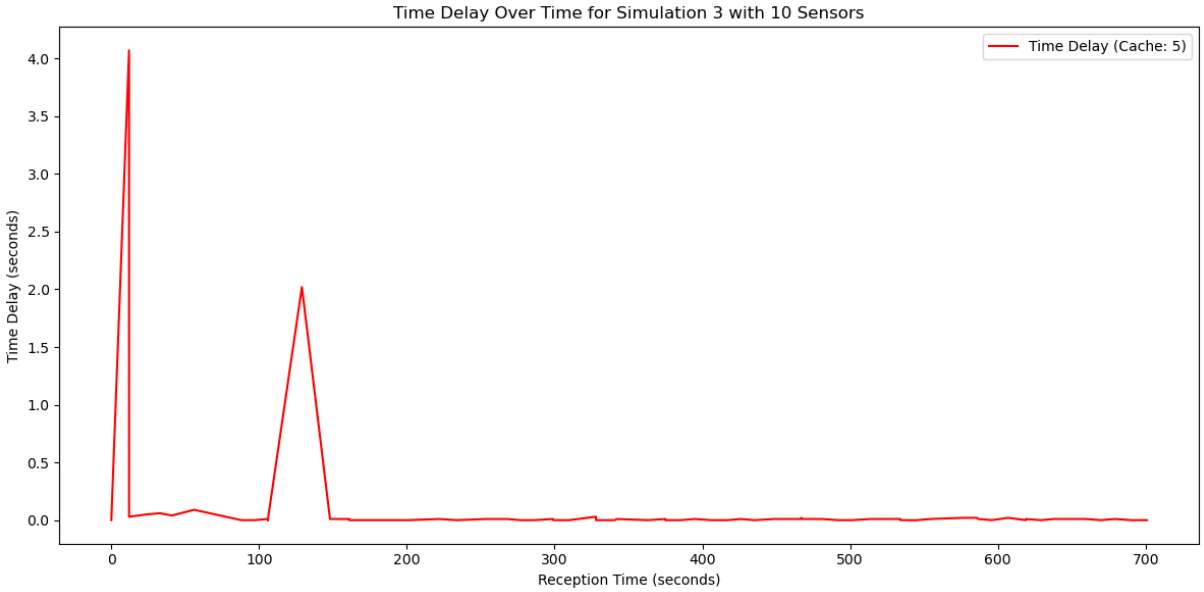


Figure 7: Simulation 3 - 10 Sensors

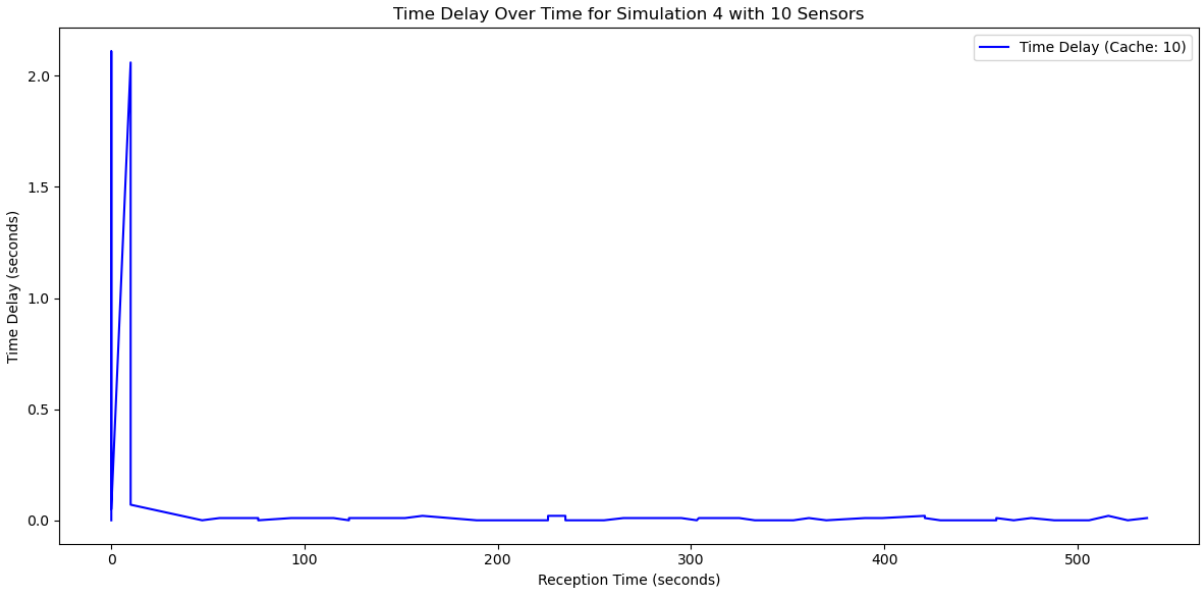


Figure 8: Simulation 4 - 10 Sensors

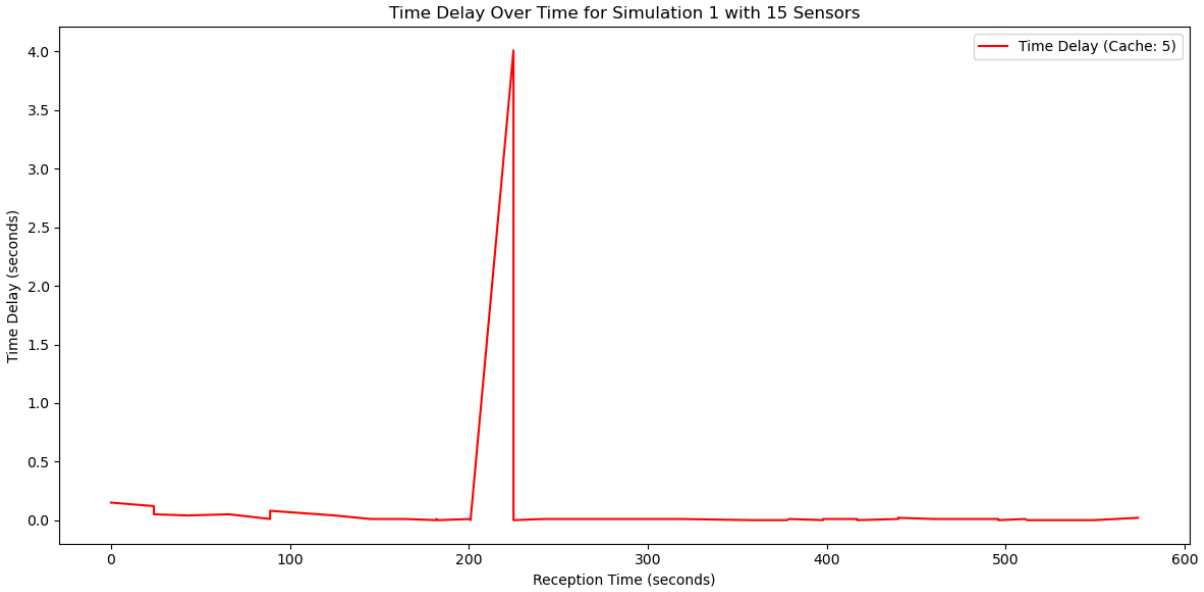


Figure 9: Simulation 1 - 15 Sensors

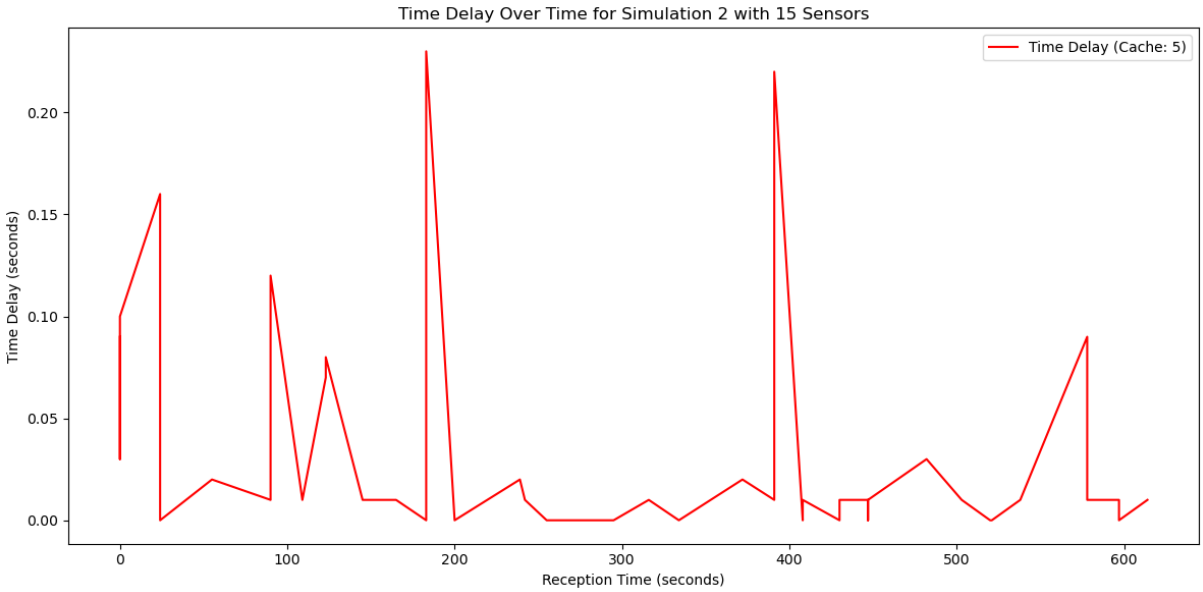


Figure 10: Simulation 2 - 15 Sensors

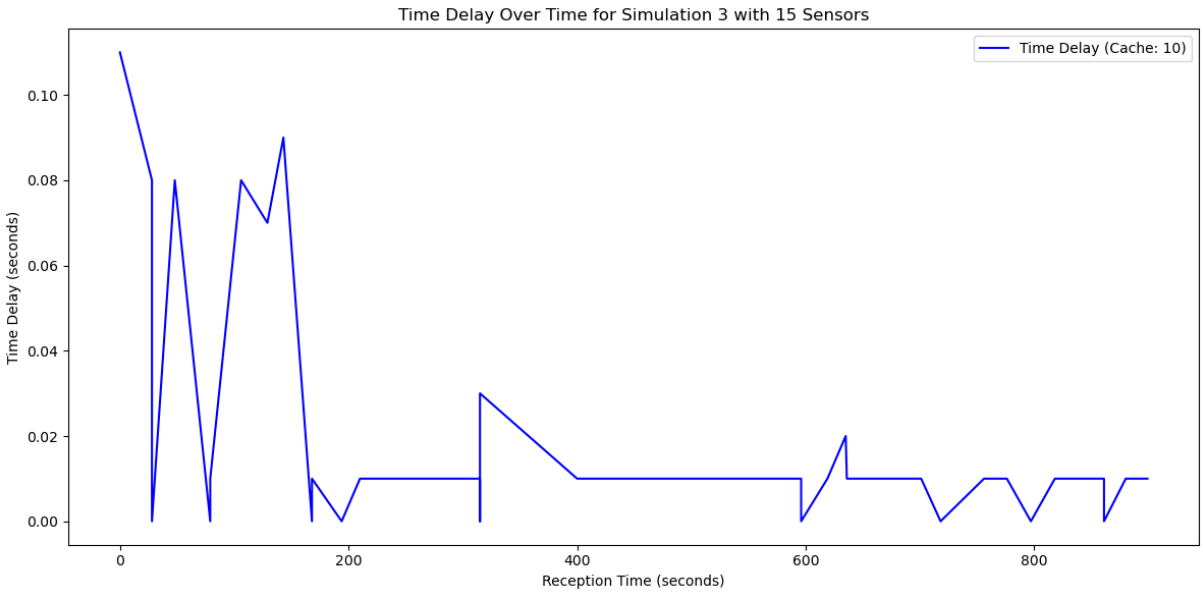


Figure 11: Simulation 3 - 15 Sensors

Considerations: As we can notice from the graphs, in the first portion of the simulations there is a considerable delay. This is due to **Gazebo** requiring some time to get the simulation going at an acceptable speed; although, when the number of sensors increased, the simulation speed was even lower and it decreases the performance even further. Because of this, in the 15 sensors scenario we experienced a high data loss

percentage due to the fact that the simulation will always run poorly as per hardware limitation. The other spikes in the graphs are due the balloons' number of tentatives to search for the data in the cache that generate delay. To obtain a more accurate results, we would need to run the simulation in a more capable machine.

4 Conclusions

In this project, we have developed a system for managing data transmission between sensors and balloons in an IoT network. The system consists of a base station responsible for sending sensors' data requests to the balloons, multiple static balloons with a cache replacement policy (LRU), and mobile sensors that patrols in a circular trajectory.

Performance evaluation of the system was conducted based on metrics such as data loss and packet delay. Multiple simulation instances were run with different numbers of sensors, and the average performance was analyzed. The results showed that the system performed well in terms of packet delay, but experienced data loss in scenarios with a high number of sensors due to hardware and software limitations.

4.1 Future directions

One possible improvement could be related to the spawning algorithm of the balloons, by considering the time complexity and the effective result. This can prove to be a challenging task, as the algorithm should be able to handle a large number of sensors and provide an optimal solution for the placement of the balloons.

The whole simulation can be rendered more complex by defining different moving policies for the mobile sensors, such as a random walk or a more complex trajectory. This would require a more sophisticated algorithm for the sensors' movement.

Another aspects that could be taken into account is varying the cache replacement policy. The latter could consider more features to assign a priority value to a specific type of data (e.g., temperature, humidity, etc.) or even to a specifc sensor device.