IoT Project 2024

Members:
- Biondi Federico 2151856, biondi.2151856@studenti.uniroma1.it
- Casciani Leonardo 2154695, casciani.2154695@studenti.uniroma1.it
- Di Paola Riccardo 2151847, dipaola.2151847@studenti.uniroma1.it

Project C

Firstly, we managed the **motion of the sensors** by making Active Sensors spawn in the simulation_launch file, instead of Sensors. Then, in order to make them move randomly, we created a function ("publish_velocity"), which makes them move only along X-axis. In particular, it checks if sensors are moving towards positive X values or negative ones and changes their direction when they get near the border of the simulation area.
To let them go all in the same direction, we handled their spawn orientation by adding the "rot" parameter in the "spawn_sdf" function of the "sim_utils" file and by giving "rot=(0,0,0)" in input to the spawning sensor function in the "sdf_launch" file.
Since they move only along X-axis, we avoided to make them spawn on the same Y value (or on the base station's Y value) by adding a list of already taken Y values in the "simulation_launch" file.
Moreover, to make published data easier to be managed, we changed the String sent to the simulation_manager by removing words and leaving only sensor id and timestamp.

Regarding **balloon movement**, we decided to distribute them along the X-axis of the simulation area (keeping Y-values equal to 0). To guarantee radio coverage to the entire set of mobile sensors, we deploy balloons in a symmetric way implemented in the "calculate_distances" function inside "fleet_coordinator" file. This function, that depends on the simulation side of the area and on the number of balloons, is then used to establish the target of each balloon in the "submit_task" function.
From the reasoning about the deployment of the balloons, we consequently found **the sensor range formula** (that can be found in the "simulation_manager" file), which is parametric with respect to the number of balloons and the simulation side of the area, and it consists in the distance from the higher balloon (after its arrival to the target position) and the worst position a sensor could reach inside the area.

To manage the limited memory that balloons have to store sensor data, we decided to implement an **LRU cache replacement policy**: ballons are subscribed to the "base_station/requests" topic and each time they see a new request from the base station a "handle_request" function is called. The latter saves the request in an array and sets a Boolean variable to true, in order to further order cache only if needed. Then, each time balloons receive sensor_data from the simulation_manager, "rx_callback" function is called. This function, firstly, orders the cache, if necessary, according to the list of requests previously saved, such that sensor_data contained in the requests list are placed on the bottom of the balloon's cache list; then, if the cache size is greater than the maximum value established, it pops the first element of the cache (the least recently used!). After this procedure, the "rx_callback" appends the new sensor_data to the list, considering it a recently used element. Finally, it publishes the cache list to the simulation_manager via the "tx_data" topic.

Furthermore, to **evict expired cache entries items** older than Δt, in the "rx_callback" is called the function "remove_old_ts" immediately before the publishing of the cache. This function slides the balloon cache and removes sensor_data with a timestamp older (which means smaller) than the threshold. The threshold Δt is updated thanks to the scheduler every X seconds, based on the kind of performances desired.

To dynamically **display balloon caches** during the simulation, we imported the dearpyguy library and implemented two functions: "init_gui" initializes the graphical elements and "update_gui" which updates displayed data when it is called, which means in the "cache_publish" function.

Regarding "**simulation_manager**" file, we decided to add subscriptions to: "balloon_{i}/tx_data" to receive cahces from each balloon through the "tx_callback" function; and "base_station/requests" to received requests from the base station, handling them with the "requests" function.
In particular, "tx_callback" function handles the arrived cache saving it in a cache dictionary, where the key is the balloon_id and the value is the the cache list.
The "requests" function calls the "check_requests_caches", which slides the caches dictionary and checks which balloons have the requested data, adding their ids to a dedicated array. All the other balloons generated a cache miss for that specific data. Finally, "requests" function publishes the information it just gathered to the base station through the topic "base_station/rx_data".

To display graphical statistics, we used dearpyguy library as well as for balloon_controller, creating again the two functions "init_gui" and "update_gui".

To handle base station requests, we created a "**base_station_controller**", through which we made the base station publish data to the "/base_station/requests" topic to send specific data requests to all the balloons via the simulation_manager; and subscribe to "/base_station/rx_data", in order to receive data from the simulation_manager through the "rx_callback" function.
To send data requests we implemented a "send_requests" function which divides the entire set of sensors in the simulation in 3 groups. One group of sensors is polled according to a **poisson distribution**, depending on the timestamps of the data produced from the sensors; another group is polled according to a **gaussian distribution,** while the last one according to a **pseudo-random distribution**. Every distribution is handled in a dedicated function and implemented thanks to the numpy library.

- The poisson distribution takes as input lambda, which represents the expected number of events occurring in a fixed-time interval. We set it to 1 to avoid requests with a timestamp too high with respect to the ones produced by the sensors until that moment.
- The gaussian distribution takes as input the mean and the standard deviation. The mean is set to a "timestamp" that is 6 seconds behind the timestamps produced up to that moment. The standard deviation is set to 2.7, in order to cover a large part of the range of the possible timestamp produced (standard deviation equal to 3 would cover the 99.7% of the interval).
- The pseudo random distribution picks a casual timestamp from a range that goes from the latest timestamps produced to 12 timestamps back in time.

Every function just discussed makes up to 3 requests per sensor and discards the ones with non-positive timestamps.

After collecting all the set of requests, the “send_requests” function publishes it to the “/base_station/requests” topic and increments the timestamp variable. The latter is used in the distribution request functions to set a sort of range of timestamps from which requests can be randomly picked. It is important to set the timestamp coherently with the frequency with which requests are sent and with the size of the balloons (an ideal scenario involves a cache size at least equal to the number of sensors multiplied by the dimension of the range from which the base stations picks the timestamps).
Then, through the “rx_callback” function, the base station receives the eventual ids of the balloons that have in cache the desired data and the cache misses generated by other balloons.

Finally, we decided to **gather statistics** related to the base station requests and these are handled inside the “update_statistics” function in the “simulation_manager”. In particular, we show the total number of requests, the number of cache misses by all balloons, the number of values found in the caches by all balloons, the percentage of satisfied requests and the percentage of satisfied requests for each group of requests (poisson, gaussian and pseudo-random).

Performance considerations:

We implemented an optimal scenario, where:

- the cache size is equal to 12 times the number of sensors (this is a lower-bound value useful to meet better performances)
- the range of timestamps considered by the base station to poll sensors is equivalent to about 12 timestamps
- the base station frequency of querying is equal to 6 seconds, with an increment of the reference instant (called just “timestamp” in the code) equivalent to 6 seconds (every 6 seconds)
- and the Δt threshold increment is scheduled every 12 seconds (in the “balloon_controller” file)

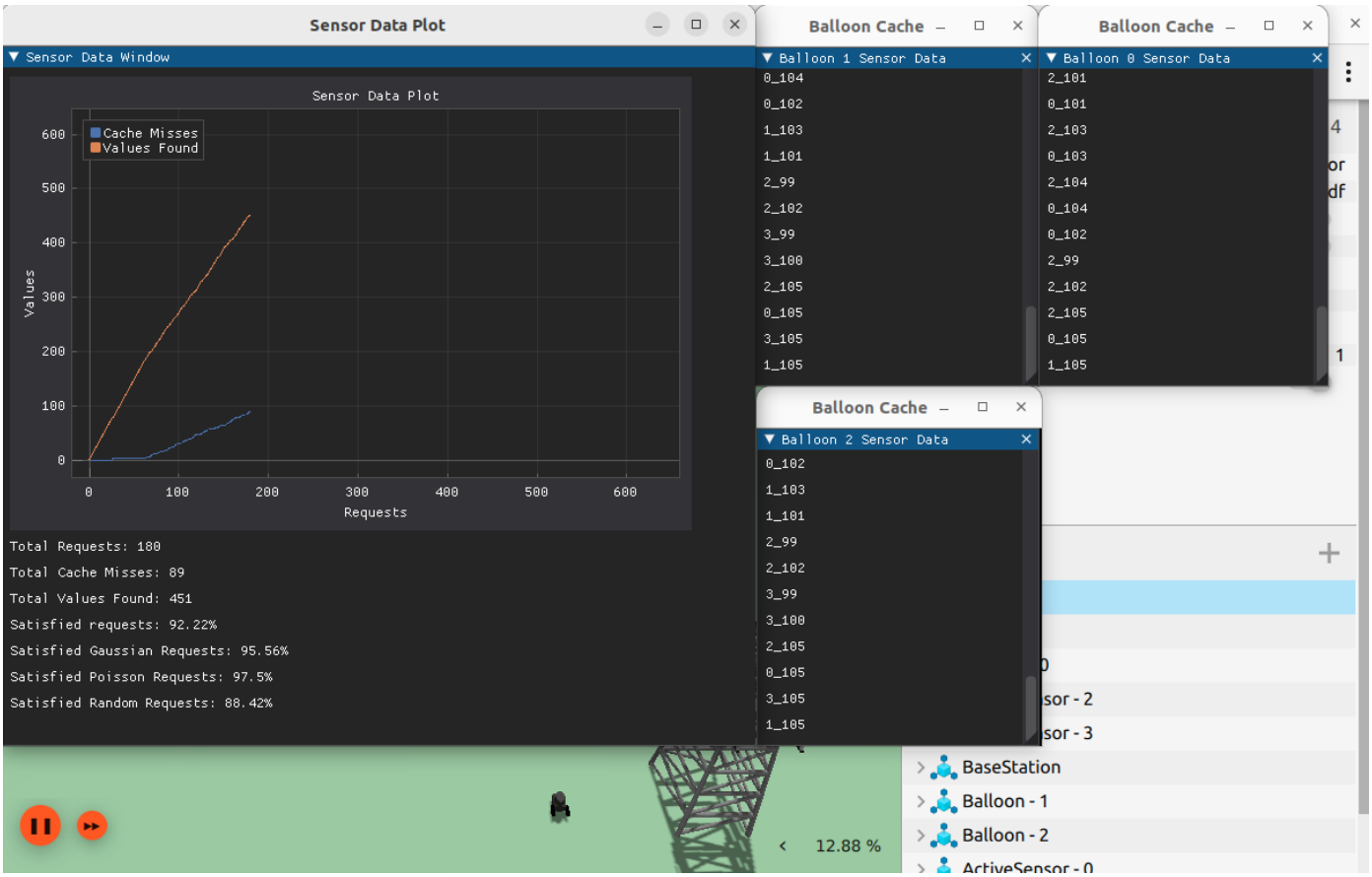
Clearly, in a realistic scenario cache could be very limited and it would be important to find a compromise. For instance, in the scenario described above the cache size could be reduced to 10 times the number of sensors, which would produce more cache misses but keep the percentage of satisfied requests high.

Also, the timestamps range from which the base station randomly picks its queries could be increased or decreased (maintaining the coherence with the other parameters described above) keeping a fixed cache size, and this would have a similar effect to only reducing the cache size.

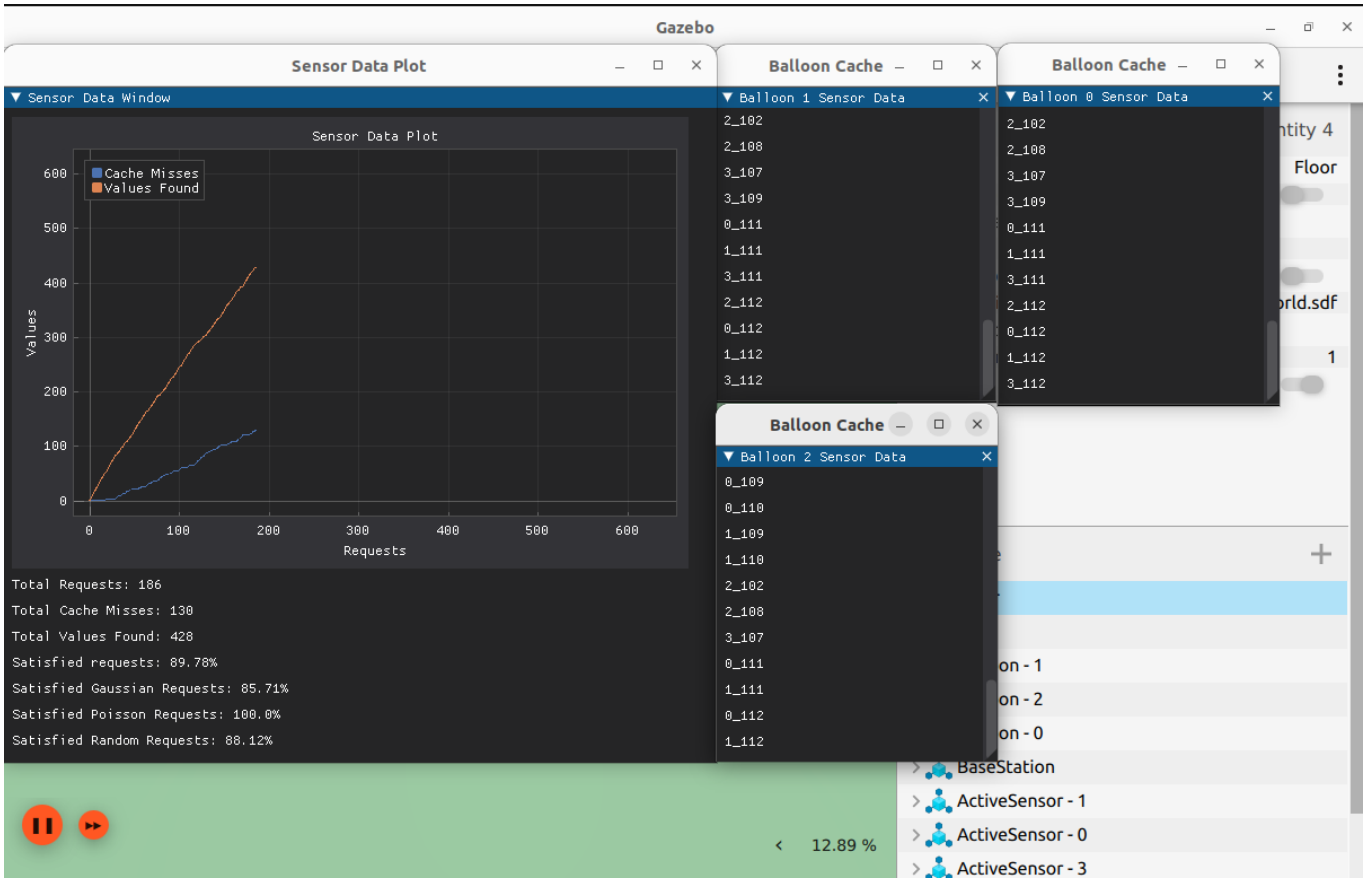
Moreover, a hypothetical scenario could involve higher cache volatility and therefore more frequent eviction of data from the cache leading to a decrease of the performances. With our implementation, even doubling the frequency of data eviction after the threshold the performances remain high.

Another critical point is the frequency related to the sending of data by the sensors. In our implementation all the sensors have the same frequency but we left commented in the code a version in which every sensor has a different frequency (based on its id) and the base station polls them according to these variations. These comments can be found in the “sensor_controller” file at the end of the “init” function, and inside each distribution function in the “base_station_controller”.

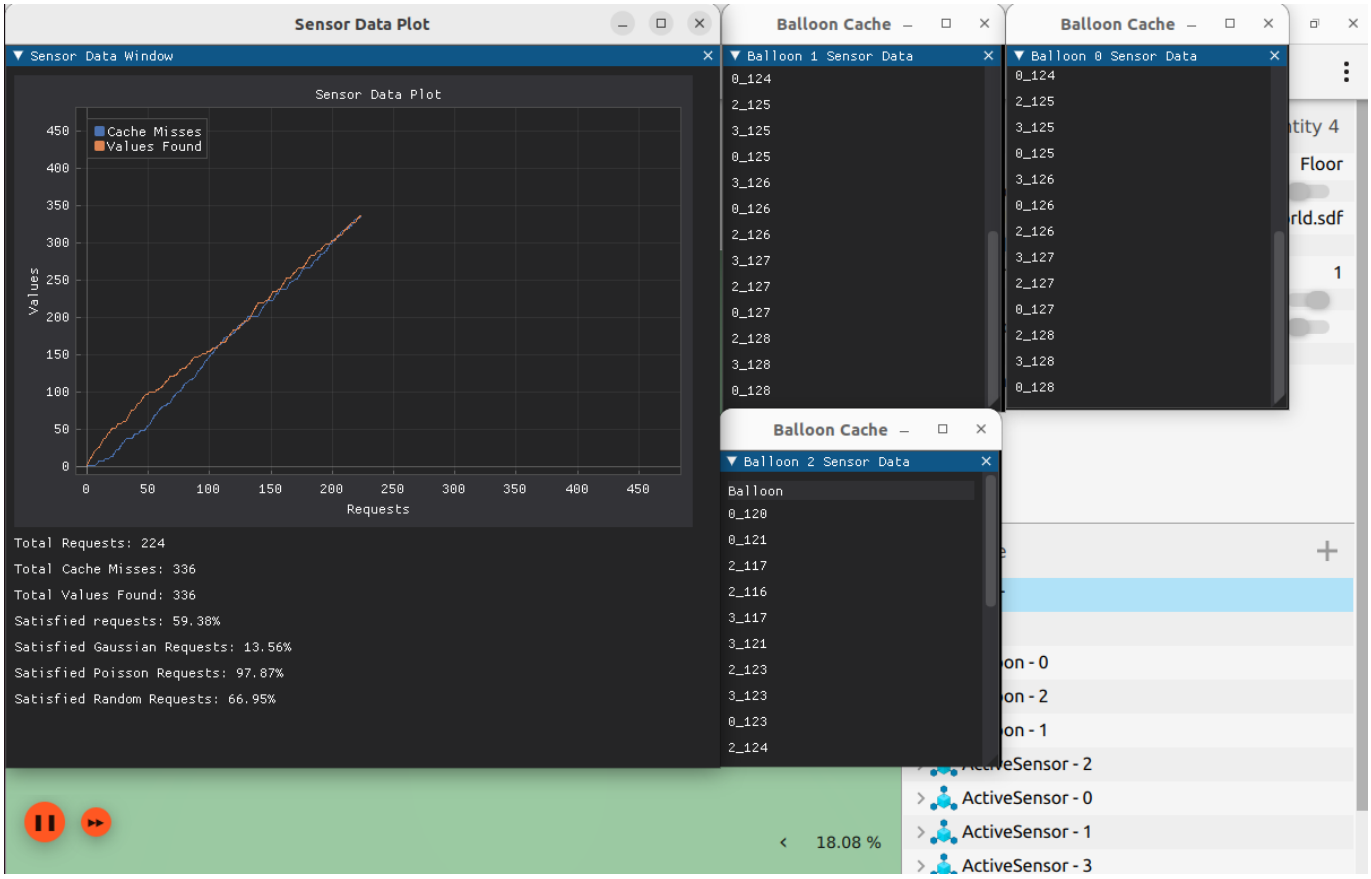
Some simulation examples:



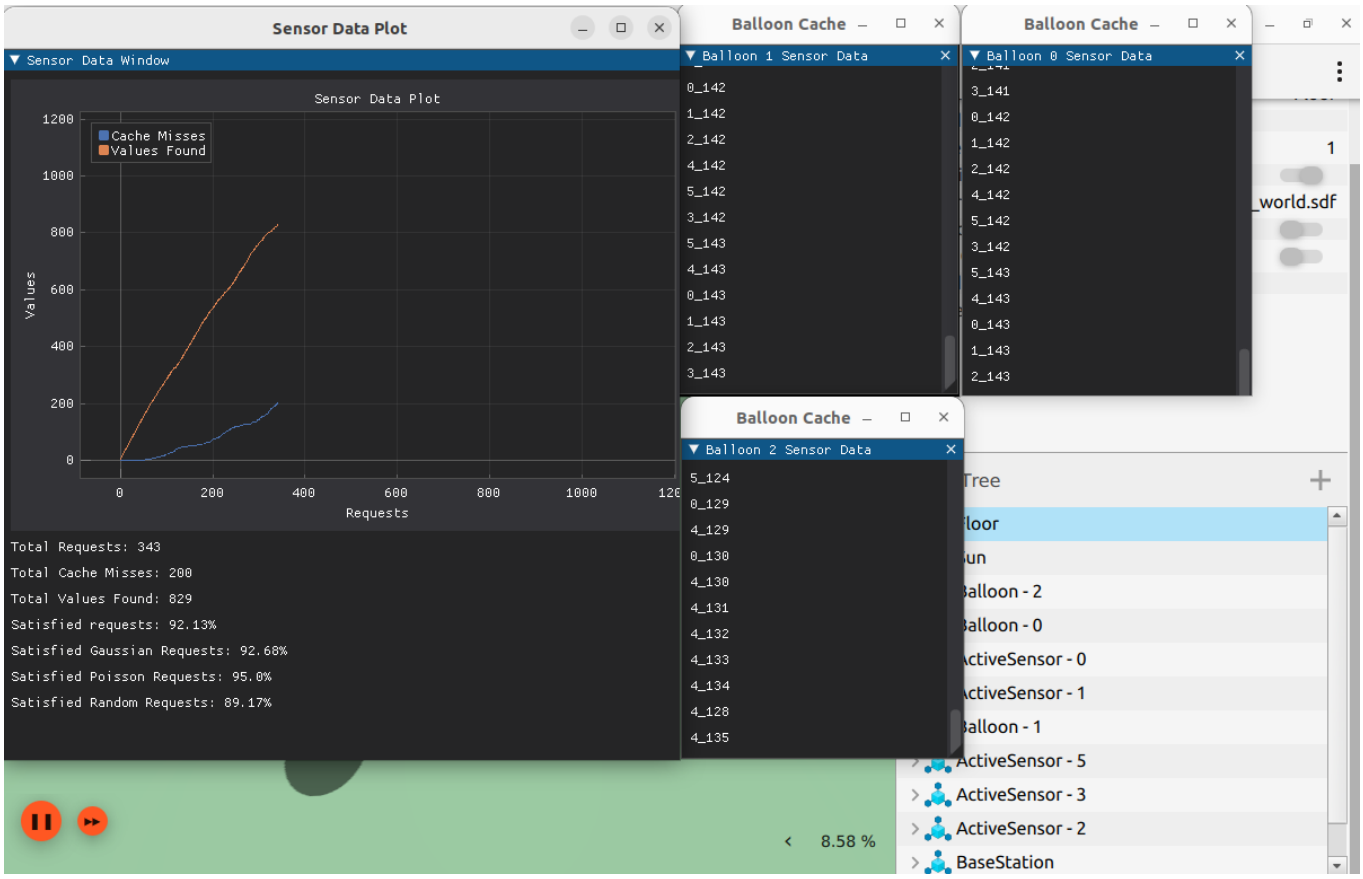
This is our implemented scenario described at the beginning of the “performances consideration” section. Number of sensors = 6; Balloons = 3; cache size = 48.



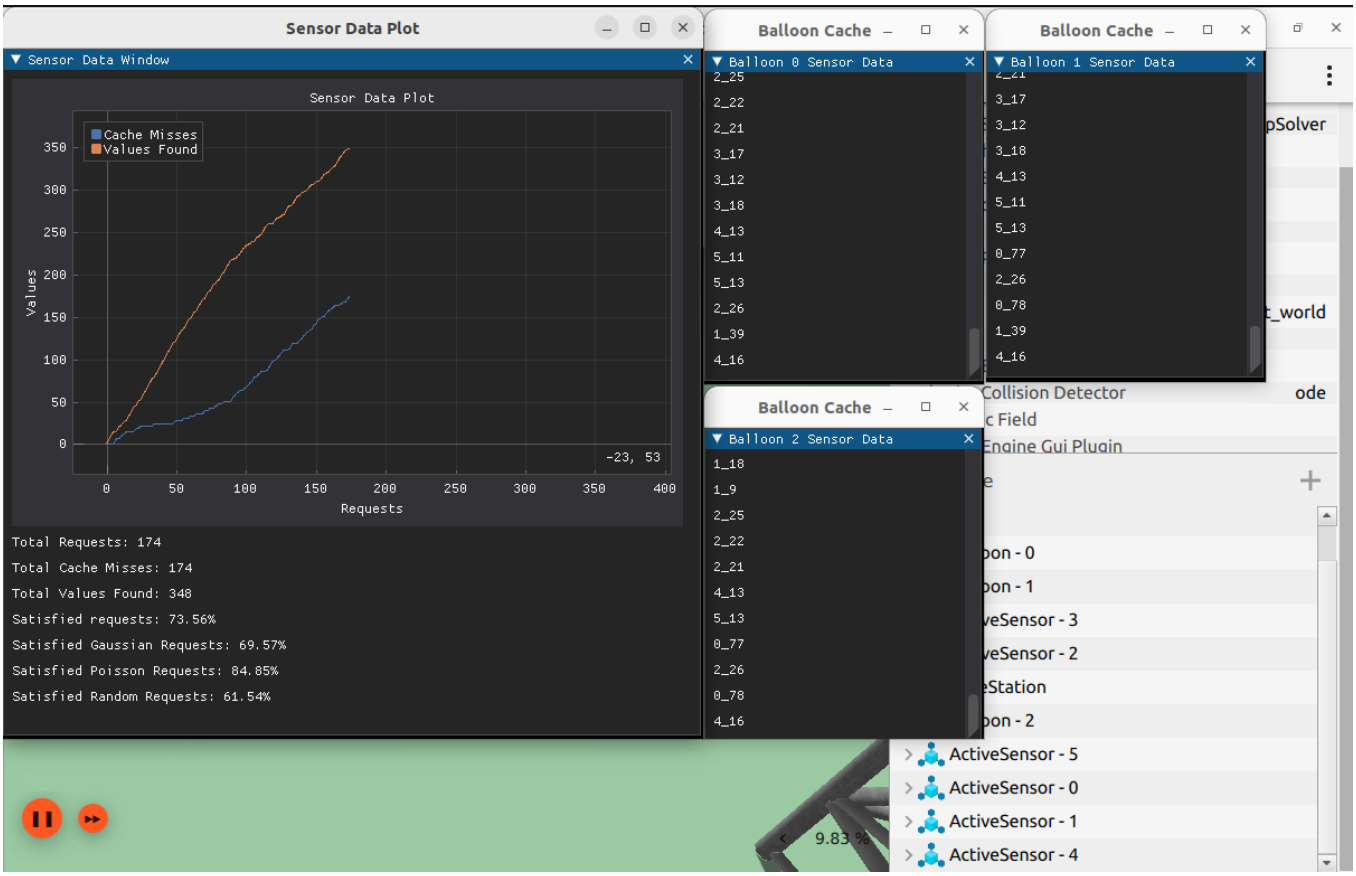
Here, the cache size has been reduced to 10 times the number of sensors. Performances are just a bit lower than the ideal scenario. Number of sensors = 4; Balloons = 3; cache size = 40.



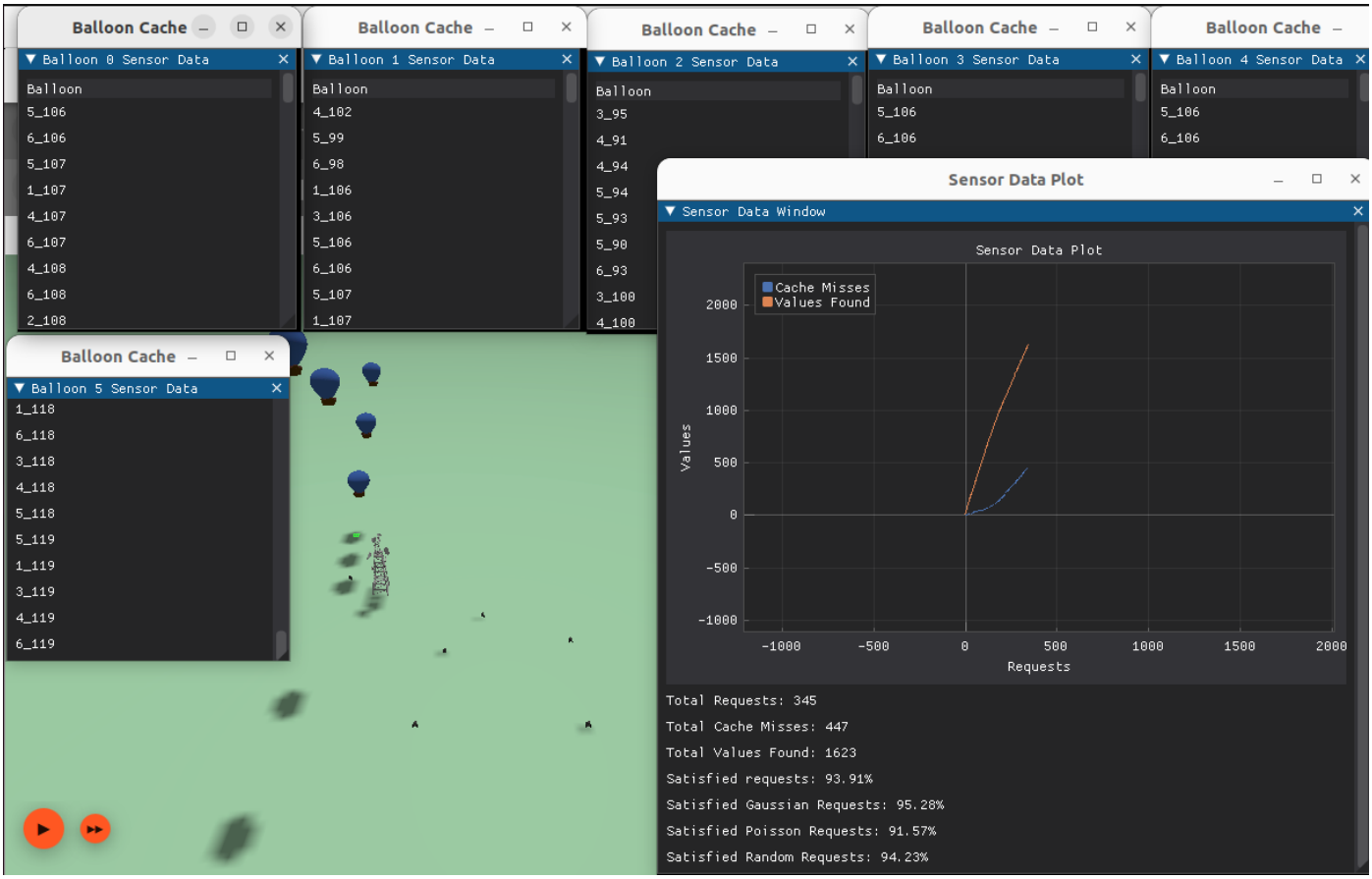
Here, the cache size has been reduced to 6 times the number of sensors (half w.r.t the ideal scenario). The percentage of satisfied requests has decreased but it's still good considering the much-limited size used. Number of sensors = 4; Balloons = 3; cache size = 24.



Here, the frequency eviction of data from the cache has been doubled but performances are still really high. Number of sensors = 6; Balloons = 3; cache size = 48.



Here, every sensor has a different rate for sending data to the balloons, and requests are performed in accordance with these rates, in a parametric way. Performances are pretty high but decreased w.r.t the ideal scenario due to the issues caused by handling different data rates. Number of sensors = 6; Balloons = 3; cache size = 48.



Here, it's again an optimal scenario but with a higher density of devices. Performances are really high. Number of sensors = 7; Balloons = 6; cache size = 84.

Final remarks:

- Some implementation details were omitted because they were left the same as the professors' initial code.
- As suggested by the professors we have created two separate launch files to avoid issues when starting the program. One contains the spawn of everything except the SDFs and the other just the SDFs.
- Other details about implementation are commented inside the code shared.
- Simulations' performances showed above are a little bit influenced by the initial stabilization phase of the balloons. In fact, at the beginning they move towards their target positions and could lose some data sent from far sensors. We left this data loss in the statistics since it's plausible in a realistic scenario a moment of adjustment.