CacheCloudNet

Edoardo Allegrini

Contents

1 Introduction

The pursuit of advancing wireless communications highlights their critical role in achieving the performance goals of future mobile communication systems. Despite significant efforts to leverage millimeter-wave (mmWave) frequencies for 5G New Radio (NR) communications, the spectrum remains scarce. This scarcity drives the necessity for enhanced spectral efficiency. An exploration of practical implementation challenges associated with increasing antenna counts in centralized systems and the interference issues linked to cell splitting indicates that the current spectral efficiency of 5G systems is unsustainable. Therefore, further innovations in cellular network architecture are deemed imperative.

Within the Internet of Things (IoT) environment, it is observed that sensing is essential for effective operation. Sensors, capable of collecting data from remote and hazardous locations, can operate without direct human oversight. However, these sensor devices generate substantial amounts of data that must be efficiently delivered and processed by a central unit to ensure seamless operations. Typically, sensors function as part of a wireless network of devices, relaying information to central units. For this network to operate effectively, it must maintain consistent connectivity. Yet, the dynamic nature of wireless networks often compromises the availability and reliability of sensing services, making continuous data offloading vital to avoid the permanent loss of critical information.

To enhance the reliability of sensing services, a novel IoT network architecture is proposed wherein communications among sensors are supported by a network of drones and balloons that provide caching and relay services. In this architecture, a central processing server connects to a base station, while several balloons serve as edge servers, facilitating data caching and multihop communications from sensors and drones to the balloons, and subsequently to the base station. The balloons are mobile devices equipped with storage, processing, and locomotion capabilities, designed to provide radio coverage over a designated area at low speeds. Conversely, drones operate within this network as highly mobile devices, facilitating message delivery from sensors either to the edge servers (the balloons) or directly to the base station.

In this specific scenario, mobile sensors operate alongside static balloons that possess limited storage capacities, capable of retaining only K pieces of data obtained from the sensors. The constraints of limited data storage and the demands from the base station necessitate intelligent caching capabilities within the balloons, which require strategic decisions about what data to cache and which to evict based on the dynamics of hits and misses during their data retrieval processes.

The challenge of managing constrained storage in a dynamic, sensor-driven environment, characterized by unpredictable data and sensor requests, presents a compelling opportunity to explore advanced caching techniques and their practical applications.

This complexity calls for the design of intelligent cache replacement policies that can effectively balance performance and responsiveness within a resource-limited system.

Furthermore, this project endeavors to extend my problem-solving skills by proposing an automatic and flexible solution for the initial deployment of balloons. The objective is to ensure that these balloons collectively provide comprehensive radio coverage for the entire set of mobile sensors, as detailed in Section 2.1.1.

2 Methodology

2.1 Initial Deployment Strategy

The initial deployment is a critical step to ensure that the entire sensor field is covered by the balloon network, while optimizing the positioning of both the balloons and the base station. The deployment is performed automatically using a dedicated class and a structured algorithm that leverages geometric properties for optimal placement. This method provides a flexible initial setup, allowing for both scalability (as the number of balloons, sensors and sensor range can be adjusted) and adaptability to different map sizes and configurations.

The code is divided into two main components:

- Balloons Placement (using a regular polygon approach).
- Sensors Placement (randomly within the balloons coverage).

2.1.1 Balloons and Base Station Placement

The core logic for deploying the balloons is implemented in the `sim_utils.py` file. It ensures that the balloons are placed in an efficient manner while maximizing coverage, based on the number of balloons and the range of the sensors. The balloons are positioned in the vertices of a regular polygon inscribed in a circle.

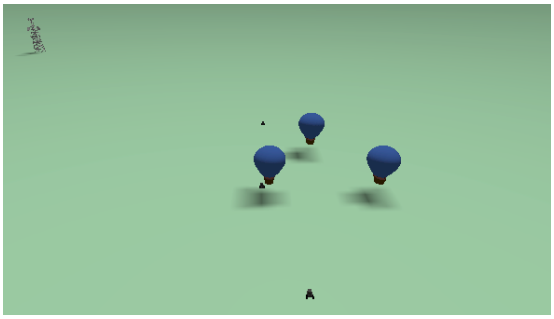
- **Base Station Placement** : the base station is placed either at the center of the map or at the edge of it, based on the launch argument provided at execution time by the user.
- **Balloons Placement**: the balloons are placed along the circumference of a circle, where the radius is calculated using the formula for a circumcircle of a regular polygon. Each balloon is positioned at equal angular intervals (using the formula angle $= 2 \times \pi \times i/\text{num\_balloons}$), ensuring that the coverage area is maximized and well-distributed across the map. Figure 1 shows a comparison of balloons placement varying the number of them; as can be seen the shape reminds a regular polygon.

- **Coverage Map**: the algorithm then computes a grid of points representing the entire map area, and checks whether each point falls within the coverage range of any balloon. This is done by calculating the Euclidean distance from each balloon to each point on the map and marking the points that are within the range of the balloon. The purpose of this step is clarified when the sensors placement and movement solutions are presented (Section 2.1.2, Section 2.5.3).

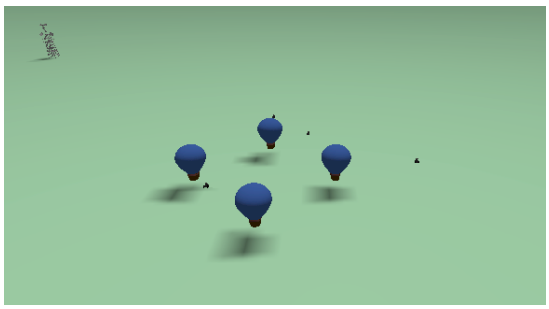
2.1.2 Random Sensor Initial Placement

The sensors are placed randomly within the covered area of the map. To ensure that each sensor is within the range of at least one balloon, the code first identifies all the points that fall within the coverage area, and then randomly selects sensor positions from this subset.

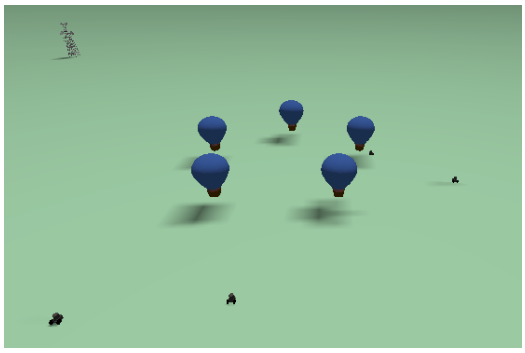
The function `get_random_points_in_coverage` selects random points from the grid that are already covered by at least one balloon. This guarantees that no sensor will be placed outside the balloon network's range.



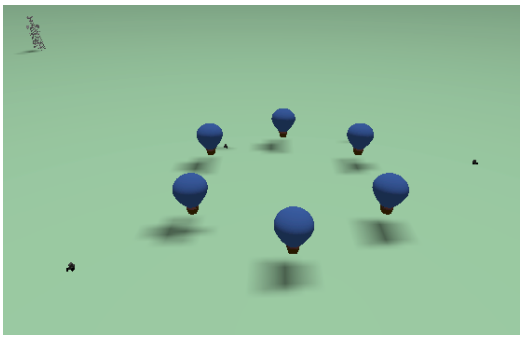
(a) Example of Initial Placement with 3 balloons



(b) Example of Initial Placement with 4 balloons



(c) Example of Initial Placement with 5 balloons



(d) Example of Initial Placement with 6 balloons

Figure 1: Example of the Automatic computation, based on number of balloons and sensors range given in input, for initial placement.

2.2 Balloon Controller

The `BalloonController` class is a critical component responsible for managing individual balloons in the distributed caching system. Each balloon acts as an intermediary between the mobile sensors and the base station, facilitating the storage and retrieval of sensor data while maintaining coverage. This class integrates message-passing, caching strategies, and action handling to ensure efficient data management and response.

2.2.1 Class Structure

The `BalloonController` class is initialized with several essential components:

- **Cache Management:** The balloon maintains a cache that stores sensor data using a specified cache replacement policy. The cache parameters such as type, size, and expiration policy are defined during initialization and are managed by the `CacheFactory` object (Section 2.7).
- **Communication Interfaces:** Several ROS 2 topics/actions/services are used to ensure the balloon interacts with the base station and the mobile sensors.

During the initialization of a balloon, an an action server is initialized to handle sensor data queries. This server listens to requests from the base station, searches the balloon's cache for the requested sensor data, and sends a response back to the base station.

2.2.2 Cache Management

The cache is configured and initialized within the `_init_cache()` method. This method declares the following parameters for cache configuration:

- `cache_type`: Defines the type of cache replacement policy (e.g., FIFO, LRU, LFU).
- `cache_size`: Specifies the maximum number of entries the cache can hold.
- `cache_expiration`: Sets the expiration time for cache entries, ensuring that outdated sensor data is automatically removed.

The cache itself is created using the `CacheFactory` Section 2.7, which builds the appropriate cache based on these parameters. The implementation ensures that sensor data is cached efficiently and retrieved when needed, minimizing data transmission delays.

2.2.3 Handling Sensor Queries

One of the key functionalities of the `BalloonController` is the ability to handle sensor data queries from the base station via the action server. The `execute_query_callback()` method processes incoming queries, retrieves the requested sensor data from the cache, and sends back a response. If the sensor data is not found in the cache or has expired, respectively appropriate error messages are caught and if needed forwarded to the base station.

```
1       def search_response(self, query_id, sensor_id, parameters):
2           basic_response = {"sensor_id": sensor_id}
3           try:
4               value = self.cache.__getitem__(sensor_id)
5               dict_converted = value.to_dict()
6               basic_response["data"] = dict_converted
7           except CacheExpiredError:
8               basic_response["data"] = "CacheExpiredError"
9           except KeyNotFoundError:
10              basic_response["data"] = "KeyNotFoundError"
11
12          response = json.dumps(basic_response)
13          return response
```

2.2.4 Receiving and Storing Sensor Data

The balloon subscribes to sensor data through the `rx_callback()` method. When new sensor data is received, it is parsed from JSON format, converted into a `SensorData` object (presented in Section 2.3), and stored in the balloon's cache (or not based on the cache policy).

This structure ensures that each balloon in the system can effectively handle its responsibilities, including maintaining accurate sensor data, managing limited storage efficiently, and responding to queries from the base station in a timely manner.

2.3 Sensor Data Class

The `SensorData` class represents real-time physiological data captured by medical sensors, typically worn by patients or troops in the field. It records and validates key health indicators such as body temperature, blood pressure, and heart rate, providing methods to check health status, log new readings, and retrieve the history of measurements.

2.3.1 Class Structure

The `SensorData` class handles three critical health metrics:

- **Temperature (°C)**: The body temperature of the individual, measured in degrees Celsius.
- **Blood Pressure (mmHg)**: A tuple representing systolic and diastolic blood pressure values, measured in millimeters of mercury.
- **Heartbeat (bpm)**: The heart rate of the individual, measured in beats per minute.

It also logs the timestamp of the measurement.

2.3.2 Real-World Scenarios

Healthcare Monitoring Consider a wearable health monitoring system that continuously tracks a patient's vital signs. This wearable device captures the patient's body temperature, blood pressure, and heart rate every few minutes and sends the data to a central system. The `SensorData` class is an abstraction of each of these measurements, ensuring that the readings are within safe and plausible ranges.

For example, in a hospital setting, multiple patients may wear such devices. A patient's sensor records:

- **Temperature:** 37.2°C
- **Blood Pressure:** (120, 80) mmHg (systolic/diastolic)
- **Heartbeat:** 72 bpm

These readings are validated and logged for future reference or real-time monitoring by medical professionals.

Military Field Application In military operations, the health and readiness of troops are critical. Troops in combat zones or remote areas may be equipped with wearable sensors to monitor their vital signs, ensuring they are physically fit for operations. The `SensorData` class can be utilized to monitor the real-time physiological status of soldiers, providing commanders and medics with immediate insights into troop health, especially under harsh or extreme conditions.

- **Monitoring Stress and Fatigue:** In high-stress situations, troops may experience elevated heart rates, abnormal blood pressure, or temperature fluctuations. The sensor data can indicate when a soldier is becoming fatigued, overheated, or dehydrated, allowing for timely intervention.

- **Remote Monitoring:** Troops deployed in forward or remote positions can have their health status continuously transmitted to a command center. Abnormal readings could trigger alerts, allowing medics to provide assistance or extract soldiers from dangerous conditions.
- **Triage and Injury Detection:** During combat or training exercises, the system can help with immediate triage by detecting soldiers experiencing critical conditions, such as hyperthermia or dangerously high heart rates, and prioritizing those needing urgent care.

For example, during a long-range patrol:

- **Soldier A's** heart rate spikes to 150 bpm and blood pressure rises to (145, 95) mmHg, indicating stress or overexertion.
- **Soldier B's** temperature drops to 35°C, possibly indicating hypothermia.

Such real-time data, collected and processed by the `SensorData` class, would alert the command center or field medic, allowing them to make timely decisions to support the troops.

2.3.3 Data Validation

The `SensorData` class includes a private method `_validate_data()` to ensure the collected data are within normal ranges for human health. If any of the readings fall outside these limits, the system raises an error. This step is crucial to avoid potential erroneous or corrupted data.

2.3.4 Health Status Monitoring

The class includes a method `check_health_status()` that evaluates the health data and provides a simple status message based on the recorded values. For instance, if the temperature, blood pressure, or heartbeat is abnormal, it returns the corresponding status.

In both healthcare and military applications, this functionality is key for triggering alerts or adjusting operational plans. For example, if a soldier shows signs of overheating or abnormal heart activity, this method would return an "Abnormal" status and could alert medics or the command team to take immediate action.

2.4 Sensor Controller

The `SensorController` class is responsible for managing each mobile sensor's behavior in the system. This includes generating, publishing, and reporting sensor data

to the network, as well as controlling the sensor's movement and responding to patrol commands. It leverages ROS 2 functionalities to manage sensor operations and integrate with other system components such as the balloon controllers Section 2.2.

2.4.1 Class Structure

The `SensorController` class is initialized with several core components, facilitating both data generation and sensor mobility:

- **Sensor Data Generation and Publishing:** Sensors randomly generate medical data such as temperature, blood pressure, and heartbeat, which are then published over a designated ROS topic.
- **Movement Control:** The sensor can patrol specified target points, moving autonomously while tracking its position and orientation.

2.4.2 Initialization and Setup

The constructor method `__init__` sets up the sensor controller by declaring essential parameters and establishing publishers, subscribers, and action servers for sensor data transmission and movement control:

1. **Data Publisher:** The `tx_topic` is a ROS publisher used to broadcast sensor data over the network.
2. **Patrol Action Server:** This action server allows external nodes (`MovementCoordinator` Section 2.5) to issue patrol commands, instructing the sensor to move to specific locations.

2.4.3 Sensor Data Generation and Publishing

Each sensor generates medical data at random intervals, simulating temperature, blood pressure, and heartbeat measurements. The `publish_sensor_data()` method is responsible for generating and publishing this data as a ROS message.

This method uses the `generate_random_sensor_data()` function to create random but realistic medical data:

- `temperature`: A randomly generated value between 35.0°C and 40.0°C.
- `blood_pressure`: Random systolic and diastolic values, simulating typical human blood pressure ranges.
- `heartbeat`: Randomly generated heartbeats per minute (bpm).

The generated data is then serialized into JSON format and published on the `tx_data` topic. This ensures that each sensor can periodically report its data to balloons which cache the sensor data for later retrieval.

2.4.4 Position Tracking

The sensor's position and orientation are continuously updated through the `store_position()` method, which subscribes to odometry messages. The method uses quaternion-to-yaw conversion to maintain accurate directional tracking:

This functionality ensures that the sensor can track its own location, which is crucial for coordinating movement during patrol actions.

2.4.5 Movement Control and Patrol Action

The sensor's movement is managed by the `Patrol` action server. The `execute_patrol_action()` method is triggered when a patrol command is received from the `MovementCoordinator`. The method controls the sensor's rotation and movement to specified target positions.

These methods ensure smooth and controlled movement, allowing the sensor to patrol its environment autonomously.

2.5 Movement Coordinator

The `MovementCoordinator` class is responsible for managing a fleet of mobile sensors within the simulated environment. It controls the sensor movements by submitting patrol tasks to individual sensors and monitors their state. The class also computes the area coverage of balloons to guide correct sensor movement.

2.5.1 Class Structure

The `MovementCoordinator` orchestrates the behavior of all sensors.

2.5.2 Initialization and Setup

In the `__init__` method, key simulation parameters are declared, and subscriptions are established to track the positions of sensors:

1. **Parameters:** The number of sensors, balloons, sensor range, and field dimensions are declared as parameters, allowing flexibility in different simulation setups.

2. **Action Clients:** Each sensor has a corresponding `ActionClient` for submitting patrol tasks, enabling asynchronous control over the fleet's movements.

3. **Position Tracking:** The class subscribes to odometry topics to monitor the real-time positions of sensors, storing this information for task submission and area coverage computation.

2.5.3 Task Submission

The primary responsibility of the `MovementCoordinator` is to keep the sensors patrolling the covered field. The `patrol_targets()` method continuously resubmits tasks to sensors that are idle.

This method ensures that each sensor is tasked with moving to a new target position once it completes a previous patrol task. The `submit_task()` method assigns the specific patrol target to each sensor. The target is chosen based on the coverage grid calculated from balloon positions.

2.5.4 Sensor States

The `SensorState` enumeration defines the possible states of a sensor:

- `IDLE`: The sensor is not currently patrolling or sensing.
- `SENSING`: The sensor is stationary and collecting data.
- `MOVING`: The sensor is in the process of moving to a patrol target.

2.6 Simulation Manager

The `SimulationManager` class orchestrates the interaction between mobile sensors and balloons within the system. The most important handling is the forwarding of sensor data to nearby balloons.

2.6.1 Class Structure

The `SimulationManager` class comprises several core functionalities essential for managing the overall simulation environment:

- **Sensor and Balloon Management:** It tracks the positions of both sensors and balloons, allowing for effective communication and data relay.
- **Data Forwarding:** The manager listens for sensor data and forwards it to the appropriate balloons based on their proximity to the sensors.

2.6.2 Position Management

The class employs the `store_sensor_position()` and `store_balloon_position()` methods to keep track of the current positions of the sensors and balloons. Notably, since the balloons must stay fixed (stable balloons do not move), the subscription on balloons odometry is destroyed as soon as the first position measurement is registered. I chose to insert this additional part to reduce the weight of the simulation making it more efficient.

2.6.3 Data Forwarding

The `forward_data()` method is responsible for relaying sensor data to the appropriate balloons. It evaluates the distance between each sensor and balloon, ensuring that only nearby balloons receive the data:

- The method calculates the Euclidean distance between the sensor and each balloon using the `math_utils.point_distance()` function.
- If a balloon is within the defined `sensors_range`, it publishes the received sensor data to its corresponding topic.

2.7 Cache Management

The `Cache` class is designed to handle data storage efficiently, allowing the balloons to manage limited memory effectively by implementing various cache replacement policies. It incorporates expiration functionality to ensure that stale data does not occupy valuable storage space.

2.7.1 Class Structure

The `Cache` class comprises several key components:

- **Cache Types:** The `CacheType` enumeration defines different cache strategies, including FIFO (First In, First Out), RR (Random Replacement), LRU (Least Recently Used), and LFU (Least Frequently Used).
- **Expiration Handling:** The `ExpiringCache` class implements functionality for expiring cache entries based on a time threshold, ensuring that old data is removed.
- **Cache Factory:** The `CacheFactory` class creates instances of the `ExpiringCache` with specific cache types and configurations.

2.7.2 Cache Types

The `CacheType` enumeration provides a structured approach to defining different cache policies. Each policy dictates how entries are managed within the cache:

- `FIFO`: The oldest entries are removed first.
- `RR`: Entries are replaced at random.
- `LRU`: The least recently used entries are evicted first.
- `LFU`: The least frequently accessed entries are discarded.

2.7.3 Expiration Handling

The `ExpiringCache` class manages cache entries with a defined expiration time:

- **Initialization:** Upon instantiation, it accepts a cache object and an expiration time in seconds. The cache is initialized using the specified replacement policy and is wrapped with expiration capabilities.
- **Item Management:** The `__setitem__()` method allows for adding items to the cache. The `__getitem__()` method retrieves items while checking for expiration:
  - If the item is found and has not expired, it is returned.
  - If the item has expired, it is removed from the cache, and a `CacheExpiredError` is raised.
  - If the item does not exist, a `KeyNotFoundError` is raised.
- **Cache Maintenance:** The class includes methods to clear the cache and check its size, ensuring effective management of stored data.

2.7.4 Error Handling

The caching system includes robust error handling to manage exceptional situations:

- `CacheExpiredError` is raised when a requested cache entry has expired, indicating that the data is no longer valid (The error will be caught by the balloon, stored and if requested forwarded to the base station).
- `KeyNotFoundError` is raised when attempting to access an entry that does not exist in the cache (The error will be caught by the balloon, stored and if requested forwarded to the base station).

2.8 Base Station Controller

The `BaseStationController` class serves as the central communication hub for managing interactions between the base station, balloons and thus sensors. Its primary responsibilities include sending queries for sensor data, receiving responses, and storing the collected data for further analysis.

2.8.1 Class Structure

The `BaseStationController` class has the following key components:

- **Parameters:** The controller initializes several parameters such as the number of balloons, sensors, cache settings, and query rates.

- **Action Clients:** It establishes action clients to interact with each balloon, enabling asynchronous communication for querying sensor data.
- **Sensor Position Tracking:** It subscribes to odometry data from the sensors to track their positions (just to appropriately know when each sensor is active and sensing and so to start the querying phase).
- **Query Management:** The class manages the querying process, including the sending of queries and handling of responses.

2.8.2 Initialization

During initialization, the `init()` method performs several actions, the important one is creating JSON file to log the configuration parameters and the results of sensor queries.

2.8.3 Query Management

The primary functionality of the `BaseStationController` revolves around managing queries to the balloons:

- **Sending Queries:** The `send_queries()` method runs in a separate thread to continuously monitor the status of previous queries and initiate new queries at defined intervals.
- **Starting Queries:** The `start_querying()` method ensures that all action servers are online and that sensors are positioned before proceeding to send queries.
- **Sending Goals:** The `send_goal()` method constructs and sends a goal message to each balloon, which includes the sensor ID and parameters to query.

2.8.4 Response Handling

The controller also manages responses from the balloons:

- **Response Callbacks:** The `goal_response_callback()` method processes the responses to the sent goals. If the goal is accepted, it waits for the result, and if rejected, it logs the information.
- **Result Callbacks:** The `get_result_callback()` method handles the results received from the balloons. It processes the sensor data and logs any errors encountered during the query.
- **Concluding Queries:** The `conclude_query()` method stores the collected results and resets the state for the next query cycle.

2.8.5 Utility Methods

Several utility methods support the main functionalities of the class:

- `compute_next_goal()`: Determines the next sensor to query and the parameters to be included in the request.
- `convert_dict_to_sensordata()`: Converts a dictionary of sensor data into an instance of the `SensorData` class for easier manipulation and storage.
- `store_result()`: Serializes the results of the queries and appends them to the JSON log file created during initialization.

2.9 Implemented Optimizations

In addition to the core functionalities outlined in the specifications, several optimizations were incorporated into the project. These optimizations address caching behavior and data polling dynamics as follows.

2.9.1 Caching Policy and Balloon Behavior

To evaluate the impact of the caching policy on balloon behavior, the following strategies were implemented:

- **Random Polling:** The base station polls the sensors at varying intervals using a random distribution, simulating real-world scenarios where sensor data is requested at different rates. This helps in testing the responsiveness of the caching mechanism under fluctuating demand.
- **Caching Strategies:** The balloons utilize advanced caching strategies (in addition to the default FIFO) such as:
  - Least Recently Used (LRU): This policy evicts the least recently accessed items, optimizing the cache's effectiveness by retaining frequently requested data.
  - Least Frequently Used (LFU): This strategy removes items that are least frequently accessed, ensuring that the most relevant data remains available for immediate retrieval.
  - Random Replacement (RR): Entries are replaced following a random eviction policy where keys are evicted in a random order.

  This approach allows performance comparisons between different caching methods, showcasing their impact on the responsiveness of data retrieval (as depicted in Section 3.4).

2.9.2 Device-Specific Expiration Thresholds

Furthermore, the implementation considers device-specific expiration thresholds ($\Delta_s t$) for sensor data. This optimization includes:

- **Dynamic Expiration Settings:** Each sensor can have unique expiration times for its data, allowing the system to adjust its caching strategy based on the importance and timeliness of the data being generated. This ensures that critical sensor information remains accessible while older, less relevant data is evicted, thereby optimizing cache utilization.
- **Improved Cache Management:** The cache is designed to effectively manage data based on these expiration thresholds, enhancing the system's ability to provide timely responses to base station queries while minimizing unnecessary cache misses.

These optimizations not only improve the overall performance of the system but also ensure that it adapts to the dynamic nature of sensor data collection and retrieval, leading to more efficient operation of the balloon-sensor network, thereby reflecting my commitment to enhancing the quality and effectiveness of the project.

3 Performance Evaluation and Testing

To comprehensively evaluate the performance of the caching strategies implemented within the balloon-sensor network, a series of tests were conducted. These tests focused on analyzing various metrics related to cache performance, including hit rates, miss rates, expiration rates, and the overall efficiency of the system. The results of these tests were visualized through a variety of plots, enabling clear comparisons between different caching strategies.

3.1 Cache Hit, Miss, and Expiration Rates

The core objective of the first set of tests was to assess the effectiveness of different caching strategies—First-In-First-Out (FIFO), Least Recently Used (LRU), and Least Frequently Used (LFU)—by analyzing the **hit, miss, and expiration rates** across various cache sizes. The testing process involved:

- **Data Collection**: Cache logs were parsed from JSON files generated during the execution of the balloon-sensor network. The relevant metrics (hit, miss, and expiration counts) were calculated for a specified range of queries.
- **Metric Calculation**: For each test scenario, metrics such as hit rate, miss rate, and expiratoin rate were computed. These metrics provide insights into how well each caching strategy performed under varying conditions.

This analysis is crucial as it helps to identify which caching strategy is most effective in minimizing data retrieval times and optimizing resource usage. A higher hit rate indicates efficient cache utilization.

3.2 Visualizations of Cache Performance

To enhance the interpretability of the results, the metrics derived from the cache logs were presented using several types of visualizations:

- **Line Plots for Hit and Miss Rates**: These plots displayed the relationship between cache size and hit/miss rates for each caching strategy. By visually comparing the performance across different configurations, it becomes evident which strategy performs best under specific cache size conditions.
- **Heatmaps**: A heatmap was created to depict the cache status over time for each balloon in the network. This visualization illustrates how cache hits, misses, and expirations occurred throughout the duration of the tests, providing a granular view of cache performance dynamics. Such representations can help identify patterns in cache behavior related to specific sensors or time intervals.
- **Bar Charts for Distribution of Hits, Misses, and Expirations**: These charts offered a clear comparison of the distribution of cache status (hits, misses, expired) for each balloon. Understanding the distribution of responses is essential for optimizing the caching policy based on actual usage patterns.

3.3 Temporal Analysis of Cache Status

In addition to assessing performance metrics, a temporal analysis of cache status was conducted:

- **Hit/Miss/Expired Rate Over Time**: This analysis involved plotting the status of each balloon's cache over the course of the testing period. By visualizing how the cache state changes over time, we gain insights into the responsiveness of the caching system and the impact of sensor data arrival rates. This temporal dimension is particularly important as it can reveal periods of high miss rates, which may correlate with increased data generation by the sensors.

3.4 Test Scenarios and Results

In the following I will present one tested scenario that I consider as one of the most meaningful. The purpose is to evaluate the caching strategies under varying conditions of the maximum cache size. I generated several plots for visualizing the scenario, in Section 3.5 there are my considerations on them.

3.4.1 Test 1: 6 sensors, 4 balloons, cache expiration 5s, cache size = 3, query rate 4s (FIFO, LFU, LRU)

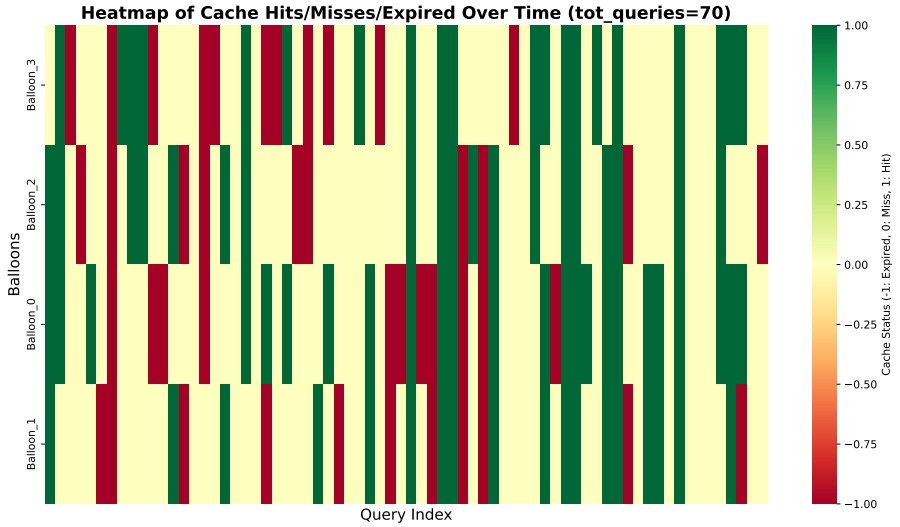


Figure 2: Heatmap for FIFO Caching Strategy

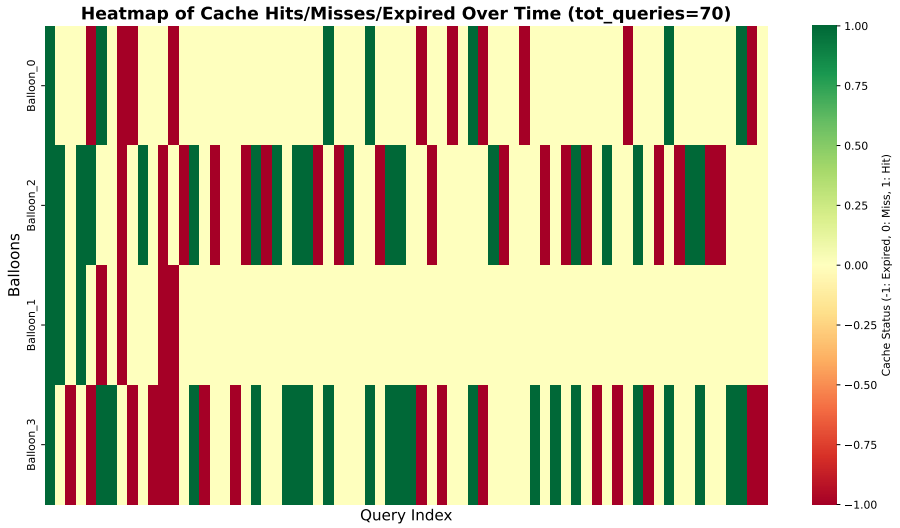


Figure 3: Heatmap for LFU Caching Strategy

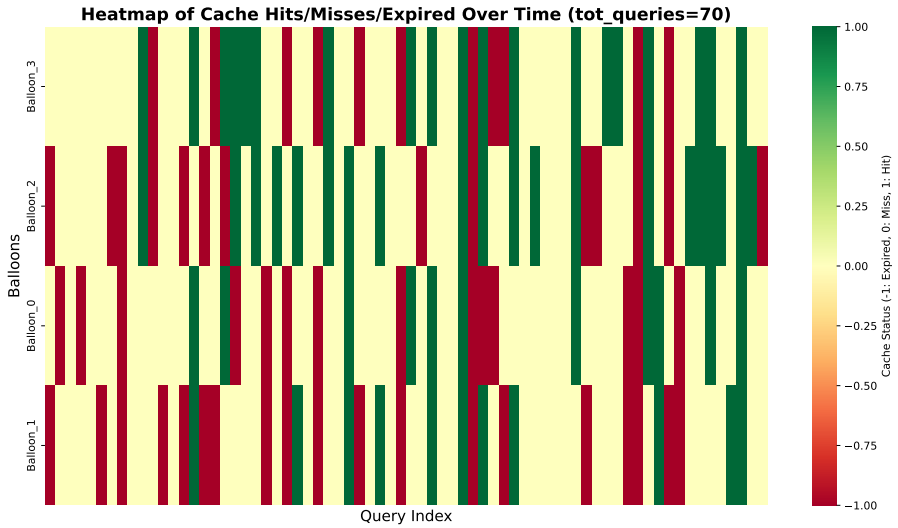


Figure 4: Heatmap for LRU Caching Strategy

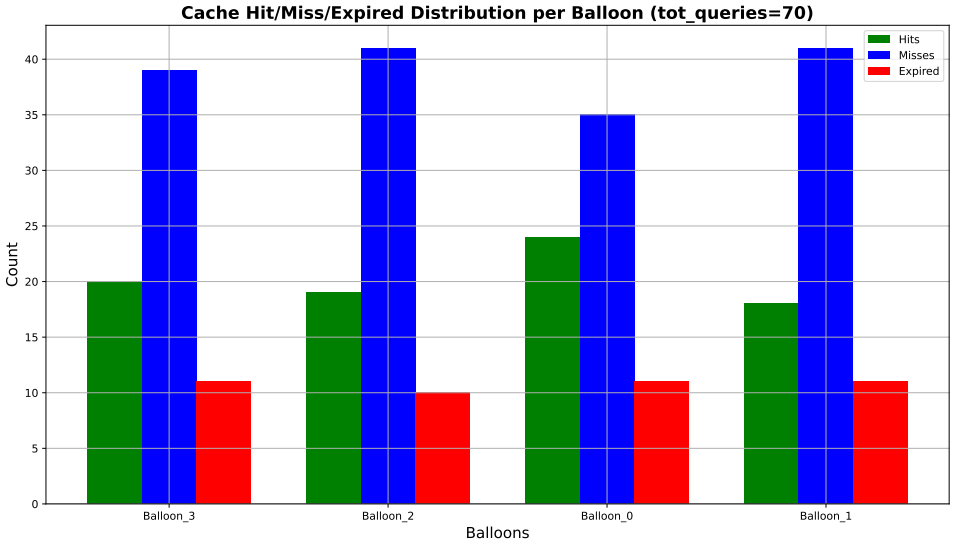


Figure 5: Hit-Miss Distribution for FIFO Caching Strategy

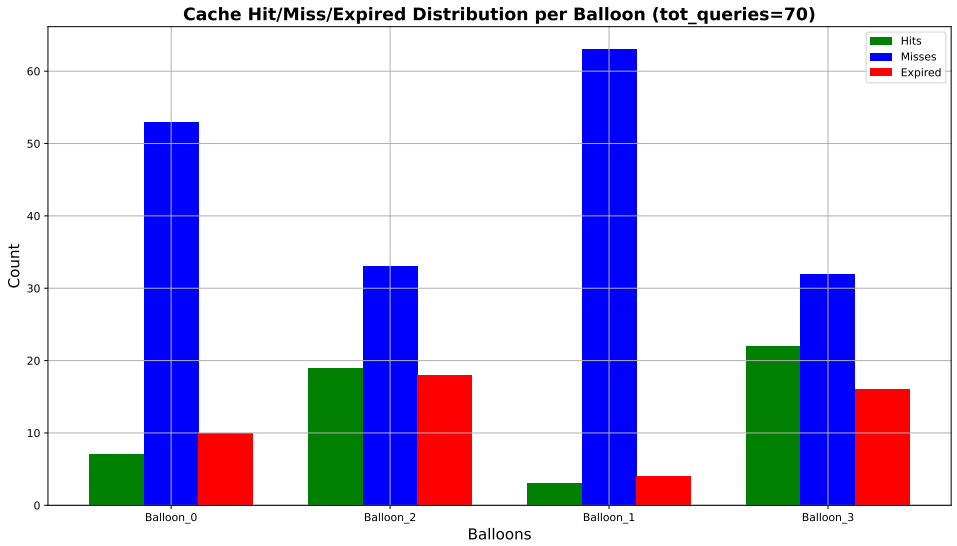


Figure 6: Hit-Miss Distribution for LFU Caching Strategy

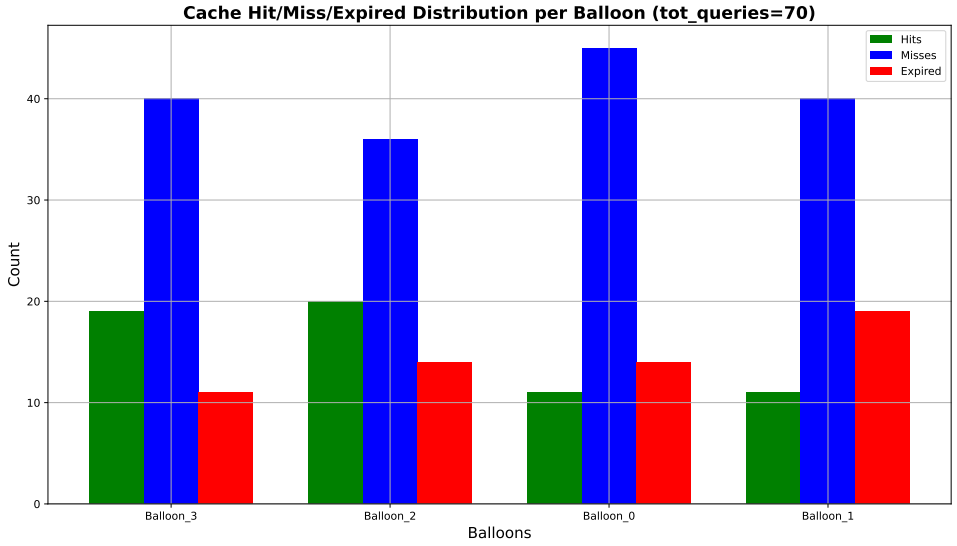


Figure 7: Hit-Miss Distribution for LRU Caching Strategy

3.4.2 Test 2: 6 sensors, 4 balloons, cache expiration 5s, cache size = 4, query rate 4s (FIFO, LFU, LRU)

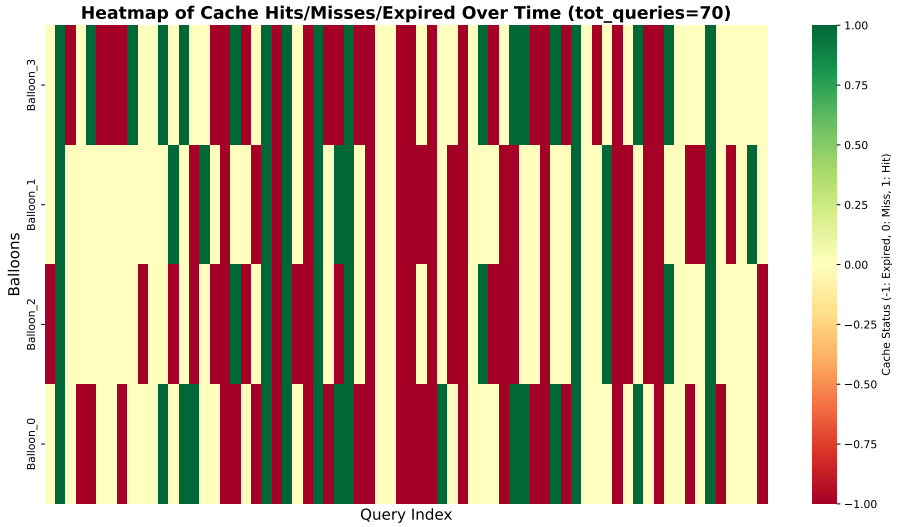


Figure 8: Heatmap for FIFO Caching Strategy

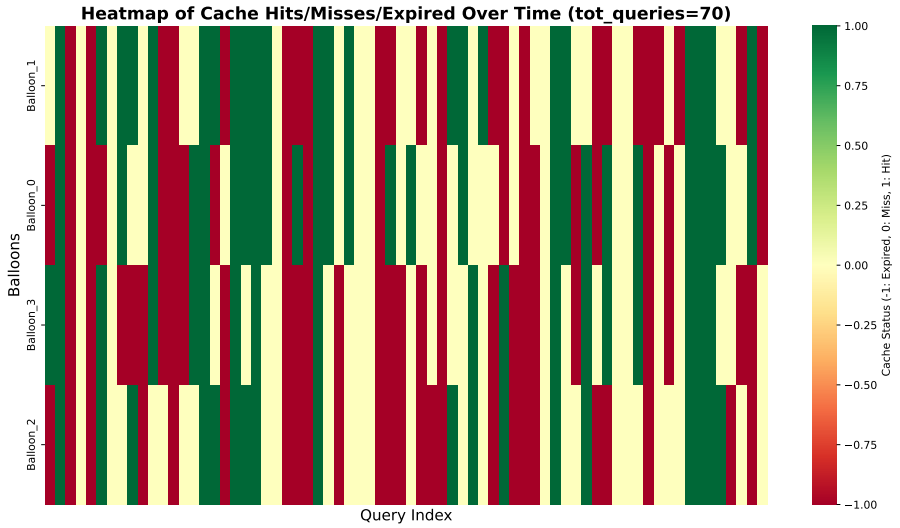


Figure 9: Heatmap for LFU Caching Strategy

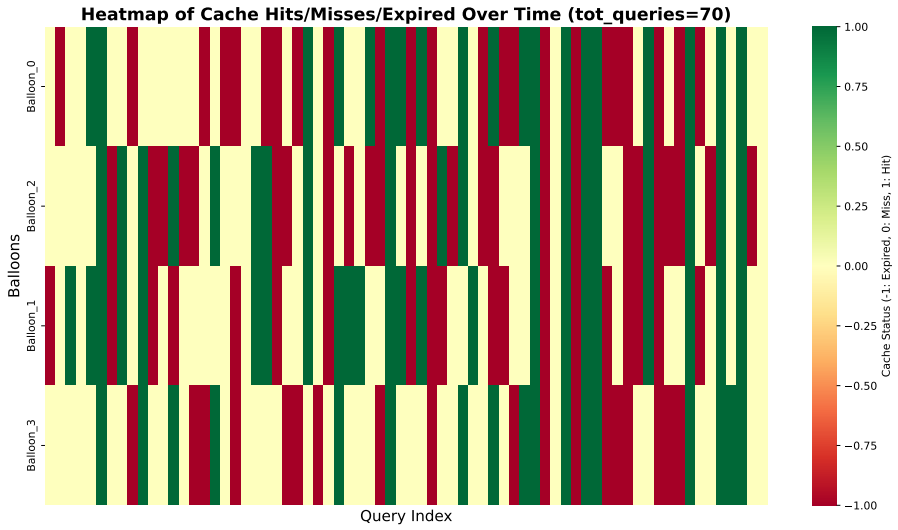


Figure 10: Heatmap for LRU Caching Strategy

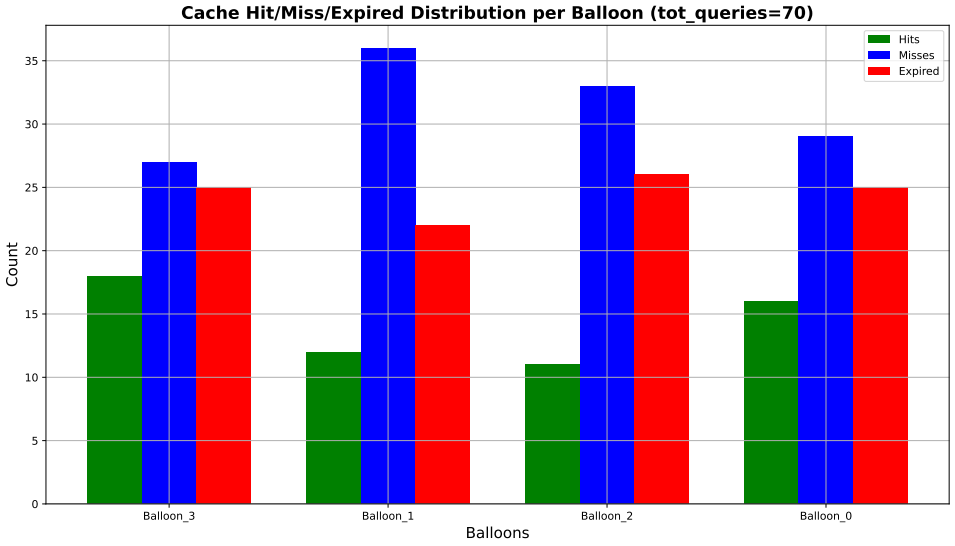


Figure 11: Hit-Miss Distribution for FIFO Caching Strategy

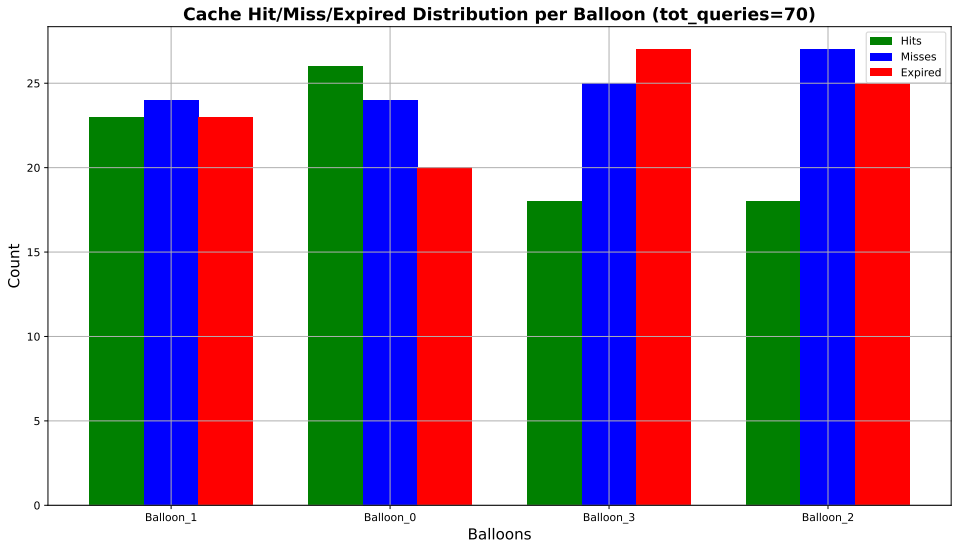


Figure 12: Hit-Miss Distribution for LFU Caching Strategy

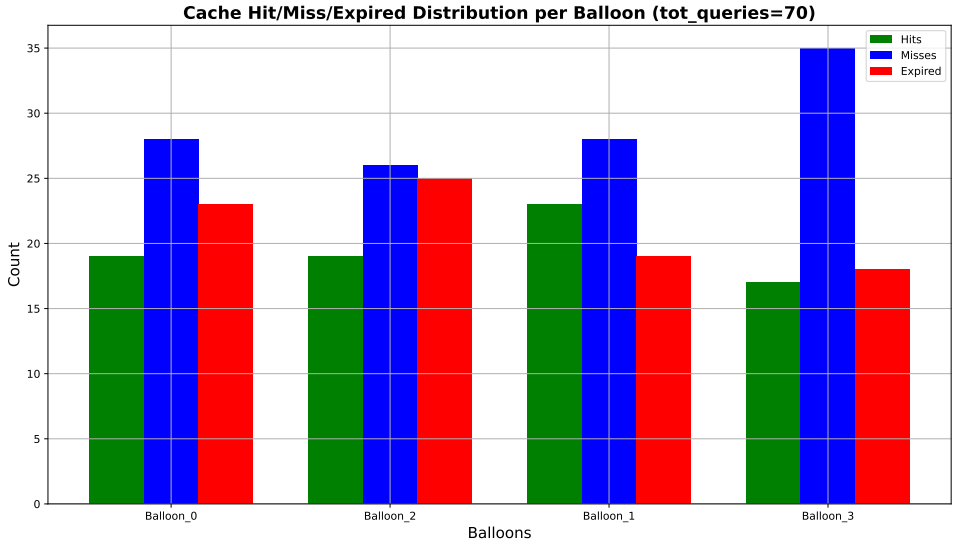


Figure 13: Hit-Miss Distribution for LRU Caching Strategy

3.4.3 Test 3: 6 sensors, 4 balloons, cache expiration 5s, cache size = 5, query rate 4s (FIFO, LFU, LRU)

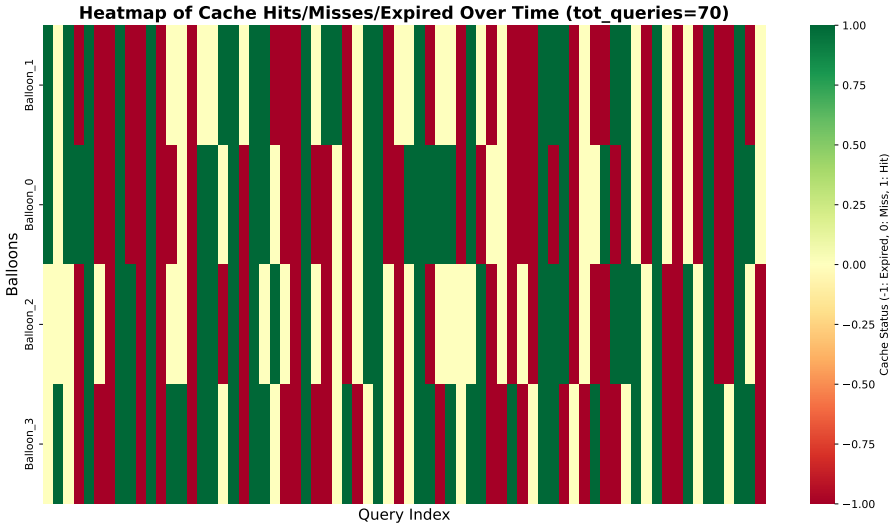


Figure 14: Heatmap for FIFO Caching Strategy

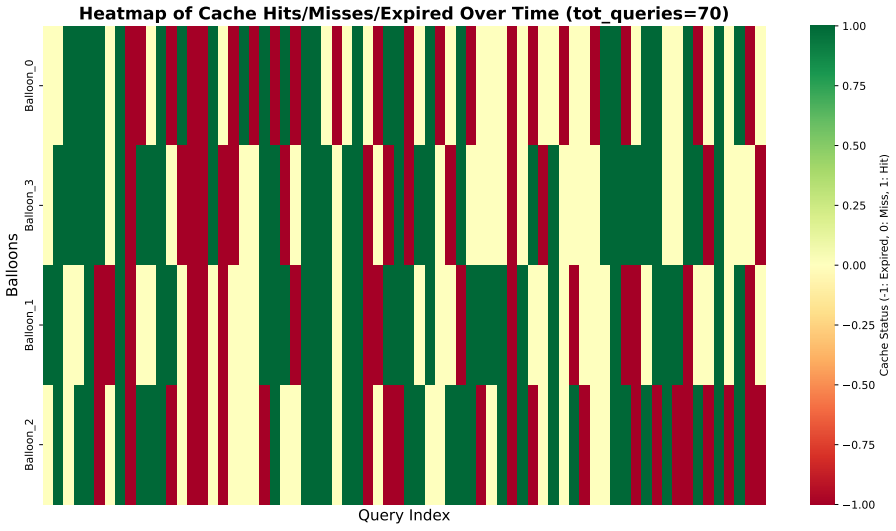


Figure 15: Heatmap for LFU Caching Strategy

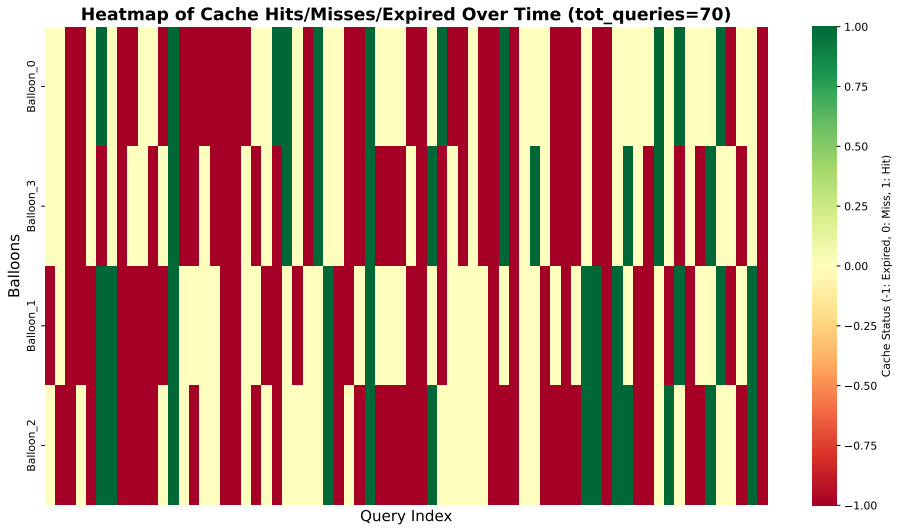


Figure 16: Heatmap for LRU Caching Strategy

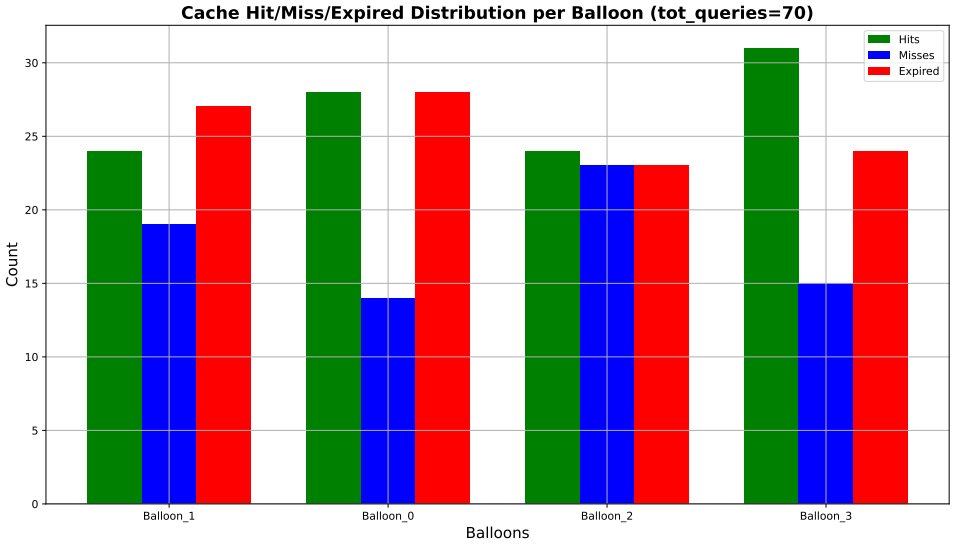


Figure 17: Hit-Miss Distribution for FIFO Caching Strategy

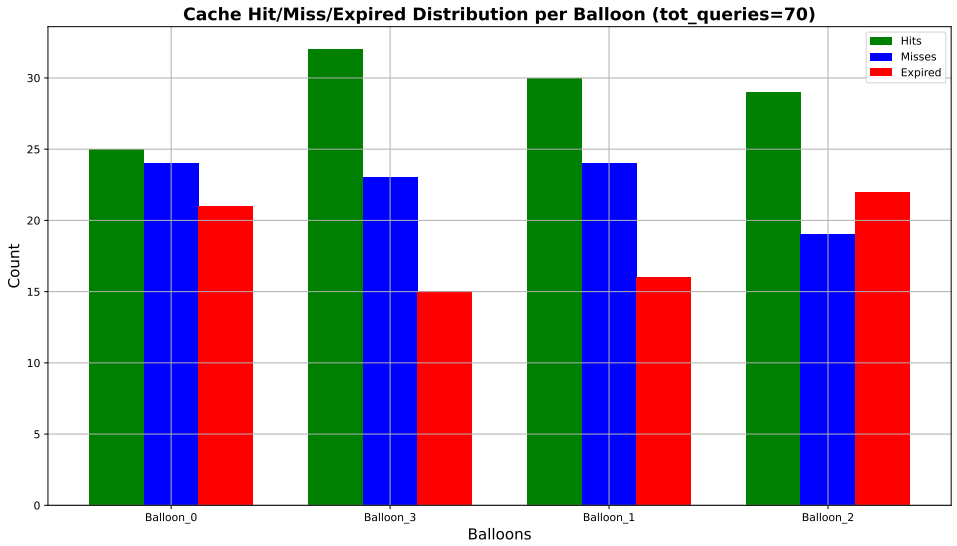


Figure 18: Hit-Miss Distribution for LFU Caching Strategy

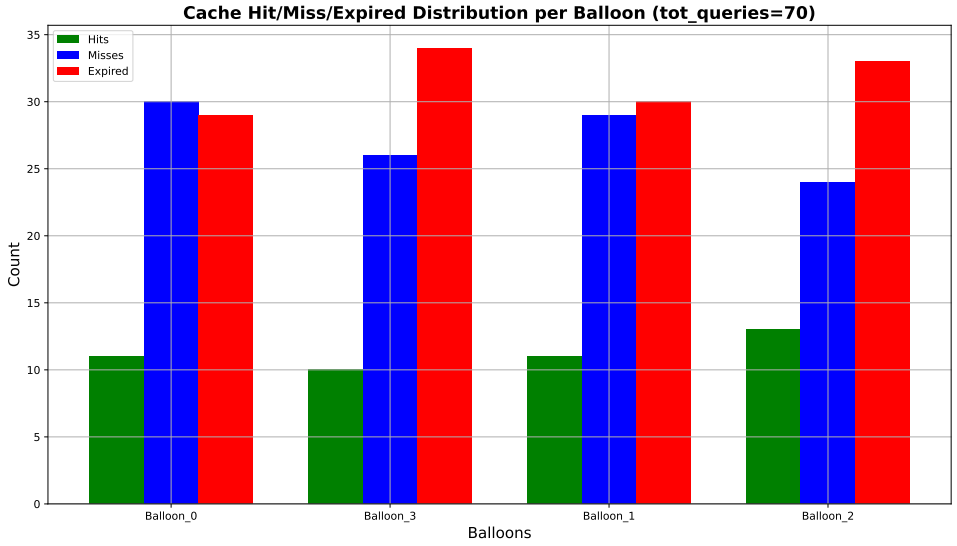


Figure 19: Hit-Miss Distribution for LRU Caching Strategy

3.5 Considerations on the Scenario

As seen in the figures above, increasing the cache size results in a progressive increase in the number of cache hits, which is a direct consequence of having more space to store previously requested data. This finding aligns with common caching principles where larger caches tend to reduce the need for re-fetching data, thus enhancing system performance.

From an academic perspective, this observation reflects the well-understood trade-

off between cache size and cache performance. Larger caches typically lead to higher hit rates, as demonstrated in this experiment. This correlation is particularly notable in scenarios where the dataset accessed is larger than the available cache size, causing frequent cache evictions and misses when the cache size is small. However, expanding the cache size mitigates this issue by accommodating more data, thus increasing the likelihood of cache hits and reducing access latencies.

The specific results in this scenario highlight the relative performance of the three caching policies: FIFO, LFU, and LRU. Unsurprisingly, LFU outperforms the other policies in this setup. The superior performance of LFU can be attributed to its ability to retain frequently accessed data, a feature particularly valuable in systems with skewed access patterns, such as sensor networks. This aligns with theoretical expectations in cache design literature that shows LFU to be well-suited for systems where the same data points are accessed repeatedly over time.

On the other hand, FIFO shows relatively stable yet suboptimal performance.

Finally, LRU's inconsistency suggests poorer performances when access frequency is more important than access recency. The sharp decline in hit rate at cache size 5 is really interesting.

3.6 Performance Comparison of Cache Policies with Varying Cache Sizes

The performance of the three cache policies—**FIFO**, **LRU**, and **LFU**—was analyzed based on **Cache Hit Rate**, **Cache Miss Rate**, and **Expiration Rate** as the cache size increases from 3 to 5. A general comparison of the results is illustrated in Figure 20, and the following key observations are what I discovered:

- **Cache Hit Rate vs Cache Size:**
  - The **LFU** (Least Frequently Used) policy exhibits a consistent increase in cache hit rate as the cache size increases. This improvement is particularly significant when moving from cache size 4 to 5.
  - **FIFO** (First-In-First-Out) starts with a relatively high hit rate at cache size 3, but its performance fluctuates as the cache size increases, showing no clear trend.
  - **LRU** (Least Recently Used) initially improves from cache size 3 to 4, but experiences a sharp decline in hit rate when the cache size increases to 5, indicating inconsistent performance with larger caches.
- **Cache Miss Rate vs Cache Size:**

- The **LFU** policy shows a significant decrease in cache miss rate as the cache size increases, reflecting its effectiveness at reducing misses with larger caches.
- **FIFO** displays a steady decline in miss rate with increasing cache size, demonstrating moderate improvement in cache performance.
- **LRU** exhibits a sharp drop in miss rate from cache size 3 to 4, but then shows an increase at cache size 5, suggesting that its efficiency decreases with larger cache sizes.

- **Expiration Rate vs Cache Size:**
  - The **LRU** policy shows a marked increase in expiration rate as cache size increases, particularly between cache sizes 4 and 5. This higher expiration rate may be contributing to its poorer hit rate performance with larger caches.
  - **FIFO** shows relatively stable eviction rates, with only small fluctuations as cache size increases.
  - **LFU** has the lowest expiration rate at larger cache sizes, aligning with its superior performance in reducing misses and increasing hit rates.

4 Conclusion

Among the three caching policies, **LFU** performs the best overall, especially with larger cache sizes, demonstrating consistent improvement in hit rate and reduction in miss and expiration rates. **FIFO** performs moderately, showing steady improvement with increasing cache sizes, though it is less efficient than **LFU**. **LRU** exhibits good initial performance but struggles with larger cache sizes, as evidenced by the decrease in hit rate and increase in expiration rate at cache size 5.

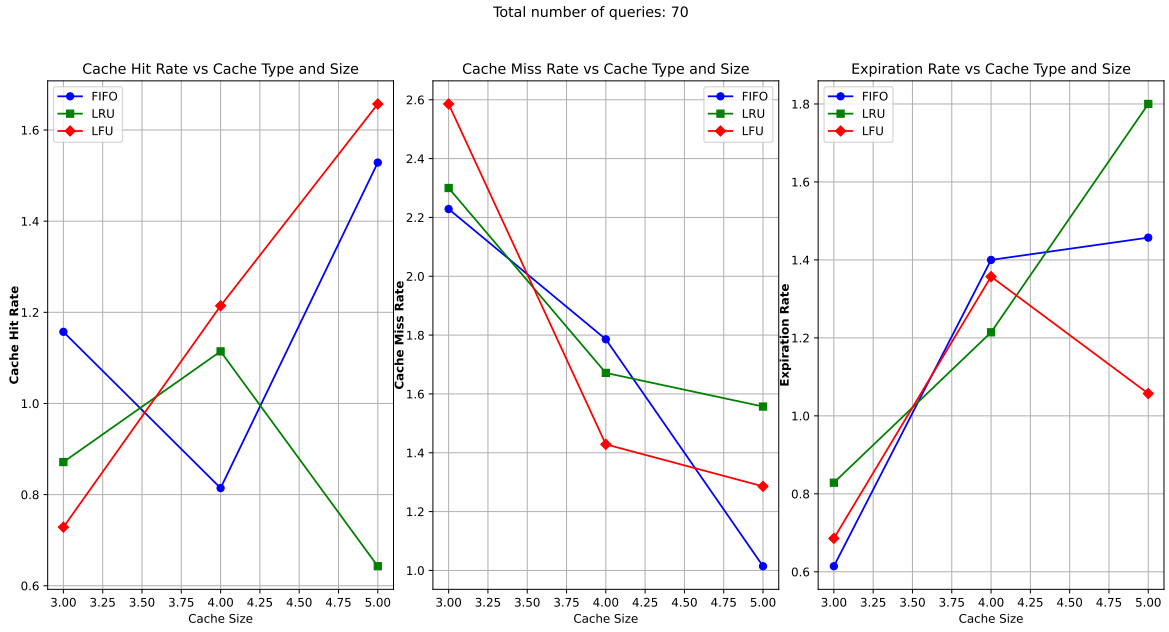


Figure 20: Comparison of Cache Hit Rate, Cache Miss Rate, and Expiration Rate for FIFO, LRU, and LFU with varying cache sizes.