Optimizing IoT Efficiency: Balancing Power Consumption and Data Accuracy Through Adaptive Polling Algorithms

Lounes Kouache, Quentin Roussel

April 2024

Course : GIN206 - Réseaux mobiles et IOT
Teacher : LIM Keunwoo

Contents

1 Introduction

In the rapidly evolving landscape of Internet of Things (IoT), sensors play an important role in the monitoring and data gathering for various applications, from environmental monitoring to smart homes. However, the very act of data acquisition—polling sensor outputs—poses a significant challenge in terms of energy efficiency. The conventional polling mechanisms, which often either over-poll or under-poll, lead to either excessive energy consumption or loss of crucial data, respectively. Over-polling unnecessarily uses the sensor's battery, while under-polling risks missing important changes in the data.

The objective of this work is to explore and establish a method to optimize the polling rate of IoT sensors that maximizes data acquisition efficiency while minimizing energy consumption. This involves developing an adaptive polling algorithm that intelligently adjusts the frequency of data requests based on the importance and variability of the information. By refining the polling process, this approach aims to preserve the sensor's battery life, enhance the longevity of the IoT devices, and ensure that critical data is captured and transmitted with optimal efficiency.

2 Polling algorithms

A polling algorithm in the context of this work is a computational method designed to determine the optimal times for activating sensors to collect data. This algorithm plays a crucial role in managing the balance between energy consumption and data acquisition efficiency. Essentially, the algorithm uses a set of predefined or adaptive rules to decide when a sensor should wake up from a low power state to measure and transmit data.

The functionality of a polling algorithm revolves around the analysis of previously collected data to make informed decisions about future data collection events. By examining past data, the algorithm can identify patterns or trends which inform the frequency and timing of future polls. The ultimate goal of the algorithm is to ensure that sensors collect enough data to maintain the quality and integrity of the information without expending unnecessary energy.

The idea of each algorithm presented is to take previously gath-

ered data and output a date, the date at which the next poll need to happen.

These algorithms also incorporate a parameter, the "target value difference," parameter defines what constitutes a significant change in sensor data. This threshold is essential as it informs the polling mechanism about when to activate, based on predicted changes in data that meet or exceed this set value. By calibrating this parameter, the polling algorithm can adapt to specific application needs, ensuring that sensors collect data only when significant variations occur.

2.1 Constant interval polling

The first method to be discussed in the context of polling algorithms is the Constant Interval Polling method. Characterized by its simplicity and predictability, it involves sampling data at fixed, regular intervals, known as the polling interval, which remains predetermined and unchanged throughout the operation of the IoT system.

In formal terms, if the last value was polled at the date t_0, the next value will be polled at the date

$$t_1 = t_0 + \Delta t \tag{1}$$

This method will be used as a benchmark for evaluating other more dynamic polling methods. Its straightforward, unchanging approach makes it an ideal baseline, allowing for clear comparative analysis against more adaptive strategies that aim to optimize energy efficiency and data relevance under varying operational conditions.

2.2 Average rate of change based polling

In many IoT applications, the data exhibit a degree of predictability, often demonstrating periodic behavior on a daily basis. For instance, an analysis of a temperature dataset may reveal that the absolute average rate of change tends to follow a distinct pattern throughout the day, peaking around 10 AM and 6 PM with lower activity levels in between. This pattern suggests that it is advantageous to sample more frequently during these peak times and reduce the frequency at night when temperature variations are minimal.

In more formal terms, consider a scenario where we are given a target temperature difference, denoted as ΔT. At a specific time t_0, a sensor records a temperature T_0. Using this information, we aim to calculate the next polling time, t_1. This calculation is based on the average rate of change of the temperature at t_0, represented as $\bar{\delta T}$. With this rate, we can estimate the time at which the temperature is expected to change by approximately ΔT using the formula:

$$t_1 = t_0 + \frac{\Delta T}{\bar{\delta T}}$$

This method requires access to a substantial amount of historical data to accurately determine $\bar{\delta T}$, ensuring reliable predictions for future sensor polling.

2.3 Linear regression based polling

Retaining the core concept from the previous method, we adapt it for situations where either limited historical data is available or the data does not display periodic behavior in terms of rate of change. However, real-world IoT systems often exhibit other forms of predictability, such as rates of change that remain relatively constant over extended periods. This characteristic can be leveraged to create a polling method similar to the average rate of change-based polling but by estimating the current rate of change using the most recent data points.

For example, using the last two data points (t_{-1}, T_{-1}) and (t_0, T_0) where $T_{-1} \neq T_0$, the time for the next poll is calculated as follows:

$$t_1 = t_0 + \left| \frac{t_{-1} - t_0}{T_{-1} - T_0} \right| \Delta T$$

However, if T_{-1} and T_0 are equal or very close, the algorithm could delay the next poll significantly. To address this issue, a maximum delay between two polls, Δt_{max}, is introduced. Thus, the formula for determining the next polling time becomes:

$$t_1 = t_0 + \begin{cases} \Delta t_{max}, & \text{if } T_{-1} = T_0 \\ \min(\Delta t_{max}, \left| \frac{t_{-1} - t_0}{T_{-1} - T_0} \right| \Delta T), & \text{otherwise} \end{cases}$$

2.4 Ideal polling

For comparative analysis, it's essential to conceptualize an ideal polling algorithm that operates with knowledge of future data values. While this algorithm is not feasible for real-world implementation due to its requirement for foresight into future conditions, it serves as a theoretical optimum against which we can benchmark practical algorithms.

The ideal polling algorithm operates under a simple premise: it initiates a poll every time the value difference reaches a specified threshold. This method ensures that data is collected exactly when needed, without unnecessary frequency or delay, thereby optimizing both energy usage and data relevance.

In formal terms, the timing for the next data collection is determined by the following condition:

$$t_1 = \min\{t > t_0, |T(t) - T(t_0)| \geq \Delta T\}$$

Here, t_0 is the time of the last poll, $T(t)$ is the temperature at time t, and ΔT is the predetermined threshold for significant temperature change. This equation seeks the minimum time t greater than t_0 at which the absolute difference between the current temperature and the temperature at t_0 exceeds ΔT. This ideal model provides a benchmark for assessing how closely practical algorithms approximate optimal data collection timing.

3 Efficiency evaluation

In this section, we will evaluate the various polling algorithms by measuring their power consumption, data accuracy, and the smoothness of data results. Power consumption is assessed by tracking the average frequency of measurements, reflecting the algorithm's impact on battery life. Accuracy is determined by comparing the differences between sampled data and actual environmental data, ensuring the algorithm captures meaningful changes without unnecessary sampling. Additionally, we will examine the "smoothness" of the data results by measuring the differences between consecutive data points, which helps in understanding how consistently each algorithm performs over time. This comprehensive evaluation will identify the most efficient algorithms in balancing energy efficiency,

accuracy, and smoothness, making them suitable for diverse IoT applications.

3.1 Evaluating the accuracy

To evaluate the accuracy of each polling method, we will calculate the average difference between the actual value at any given time and the value last sampled by the method. Denote $(t_i)_{i\in[0,N]}$ as the strictly increasing sequence of dates at which values are sampled, and let $f : \mathbb{R} \to \mathbb{R}$ represent the continuous function that models the data being sampled. The average error can formally be defined as:

$$\bar{\varepsilon} = \frac{1}{t_N - t_0} \sum_{i=0}^{N-1} \left(\int_{t_i}^{t_{i+1}} |f(t) - f(t_i)| \, dt \right)$$

However, the dataset we are working with consists of a sequence of discrete points sampled at a high rate, which we will sub-sample using the methods previously described, rather than a continuous function. Thus, we need to adapt our definition of error rate accordingly. If we denote $(t_i)_i \in [0, N]$ as the sequence of dates at which our data points are sampled to form the dataset, and $(t_{i_0}, t_{i_1}, \ldots, t_{i_p})$ as the dates at which values are actually polled by the algorithm, we can define the average error similarly:

Define $\phi(k)$ for $k \in [0, N]$ as:

$$\phi(k) = \max_{\substack{j\in[0,p] \\ i_j \leq k}} i_j$$

With this definition we have $t_{\phi(i)}$ the date of the most recently sampled point, which is the last known value at date t_i. The average error is then:

$$\bar{\varepsilon} = \frac{1}{N+1} \sum_{i\in[0,N]} \left| f(t_i) - f(t_{\phi(i)}) \right|$$

This adapted measure provides a practical way to quantify the deviation between the sampled values and the actual data points in a discrete setting.

3.2 Power efficiency

The power efficiency of each method is gauged by calculating the average duration between two successive polling events. This duration is inversely proportional to the average power consumption of the system, meaning that longer intervals between polls indicate lower power usage.

4 Results

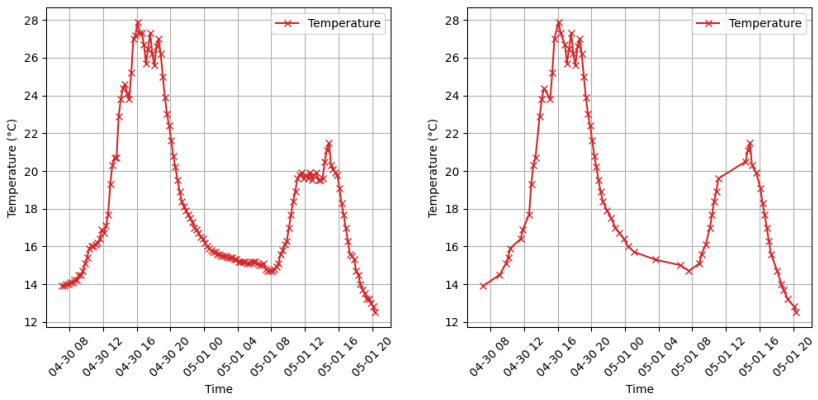
To evaluate the previously described methodologies, we utilize a dataset comprising temperature records from a greenhouse, collected daily over a period exceeding four years. The chosen dataset is exemplary for testing due to its inherent daily cyclic behavior, punctuated by rapid temperature fluctuations and prolonged periods of slow change. It is important to emphasize that the techniques discussed should not be deployed for applications where data accuracy is crucial. There is a risk of losing significant information, such as brief peaks in data that return to baseline before the next data collection point, or constant values that suggest less frequent polling but may miss important fluctuations. These methods are, however, appropriate for use with energy-constrained sensors that are not monitoring critical information, allowing for efficient data management without comprehensive data capture.

4.1 Example of polled data

The algorithms were implemented in Python and evaluated using a dataset comprising 150 data points, equivalent to 37.5 hours of recorded data. To calculate the average rate of change across the entire dataset, we first determined the rate of change between each consecutive data point. Subsequently, these rates were categorized by the hour of the day, and an average was computed for each hour. The resulting data is illustrated in Figure 3.

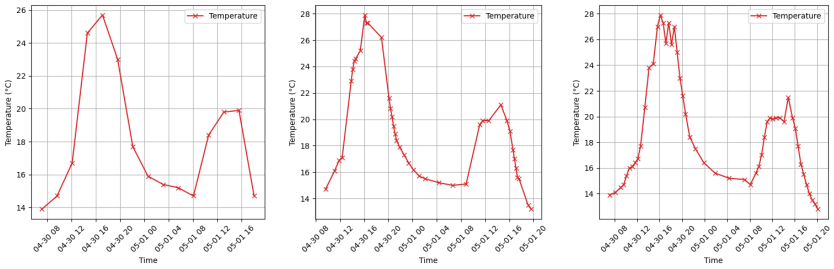
Figure 2 demonstrates that the constant interval and linear regression-based polling methods fail to capture the rapid temperature fluctuation observed at 4 PM on April 30th. In contrast, the average rate of change-based method accurately detects this fluctuation. However, it leads to oversampling on the following day at 12 AM when

compared to the other methods.



(a) (b)

Figure 1: (a) Original data: A 37-hour segment of the greenhouse temperature dataset, showcasing typical daily fluctuations. (b) Optimally sampled data: The same 37-hour segment using a temperature change threshold of $\Delta T = 0.5$°C for data collection.



(a) (b) (c)

Figure 2: (a) Data sampled at a constant interval of 150 minutes, demonstrating the risk of missing significant temperature changes. (b) Data sampled using a linear regression based method, maintaining a temperature change threshold of $\Delta T = 0.5$°C. (c) Data sampled based on the average rate of change, with $\Delta T = 0.5$°C.

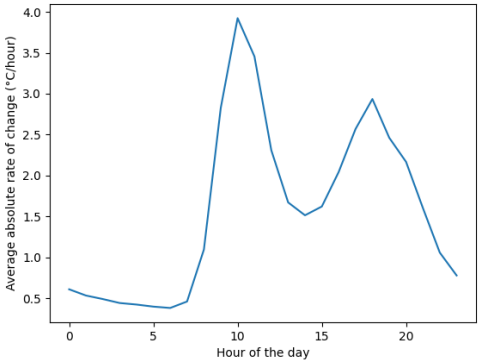


Figure 3: Hourly average absolute rate of change of temperature throughout the dataset, highlighting time-specific variability and peak change periods.

4.2 Statistical analysis

To quantitatively assess the efficiency of the methods, we apply the metrics defined in Section 3 to calculate the average error and the average interval between two polls, adjusting the ΔT parameter. The resulting data is illustrated in Figure 4, which indicates that both the linear regression and average rate of change based polling methods outperform the naive method.

For example, the constant interval polling method requires an average of 2063 seconds between polls to achieve an average error of 0.5°C, whereas the linear regression method reaches this accuracy with an average interval of 2560 seconds, and the rate of change based method does so with an interval of 3051 seconds, compared to an optimal interval of 4030 seconds. These intervals correspond to battery life improvements of 124%, 148%, and a maximum theoretical gain of 195%, respectively.

It is evident that the average rate of change method consistently performs better in our scenario; however, it's important to note that this method depends on having access to previous data and a periodic rate of change, unlike the linear regression method, which does not require such conditions.

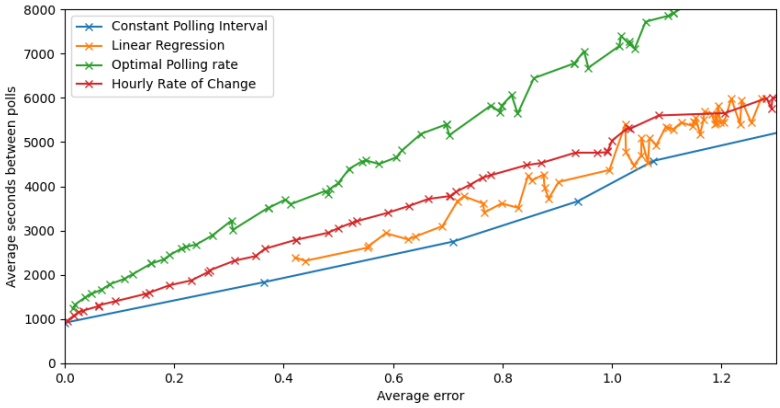


Figure 4: Comparative efficiency analysis of polling methods based on average error and polling interval with varied ΔT settings

It should be noted that the dataset used in this analysis contains samples taken every 900 seconds, which sets a lower limit on the polling rate. When the calculated time for a poll falls between two existing samples, the subsequent sample is selected.

5 Practical implementation

In our study, we utilized theoretical values from a dataset to initially calibrate the polling intervals for real-time implementations, detailed in Algorithm 1 (linear regression based polling) and Algorithm 2 (hourly average rate of change based polling). These algorithms are designed to dynamically adjust the frequency of data collection based on historical patterns, thus optimizing energy efficiency.

Adaptable to real-time data processing, these algorithms are integrated with an API that fetches live sensor data, enabling them to respond to changing conditions and push updates to a ThingsBoard API for visualization. The Python code for these algorithms, which facilitates their deployment in real-time systems and the visualization of data trends through ThingsBoard, is available our GitHub repository.

6 Conclusion

In this work, we have explored various polling algorithms designed to enhance the efficiency of IoT systems by carefully balancing power consumption and data accuracy. Our approach combined theoretical insights with practical implementations to address the crucial trade-offs inherent in IoT data collection.

Our initial models used theoretical values from a dataset to establish baseline efficiencies for different polling strategies. These models helped us understand how variations in data collection frequency could impact energy usage and data integrity, which are critical for sustainable IoT operations.

We then progressed to real-time implementations, detailed in Algorithm 1 (linear regression based polling) and Algorithm 2 (hourly average rate of change based polling). These adaptations allowed the algorithms to interact with live data streams and demonstrated their potential in practical scenarios such as environmental monitoring and smart home infrastructure, as shown with the example of a temperature sensor in a greenhouse.

Algorithm 1 Linear Regression Based Polling Algorithm

Initialize: *SENSOR_API_URL* as URL to fetch sensor data
Initialize: *THINGSBOARD_API_URL* as URL to post data to ThingsBoard
Initialize: *TARGET_DIFFERENCE* as desired temperature change threshold
function GetSensorData
    Fetch data from *SENSOR_API_URL*
    Extract *timestamp* and *temperature* from the data
    return *timestamp*, *temperature*
end function
function PostToThingsBoard(payload)
    Set headers to {'Content-Type': 'application/json'}
    Post *payload* to *THINGSBOARD_API_URL* with headers
    return HTTP status code of the post request
end function
last_time, *last_temp* ← GetSensorData
while true **do**
    current_time, *current_temp* ← GetSensorData
    time_diff ← *current_time* - *last_time*
    temp_diff ← *current_temp* - *last_temp*
    if *temp_diff* = 0 **then**
        temp_diff ← very small number
    end if
    average_rate_of_change ← absolute(*temp_diff* / *time_diff*)
    next_interval ← absolute(*TARGET_DIFFERENCE* / *average_rate_of_change*)
    payload ← {'temperature': *current_temp*}
    PostToThingsBoard(payload)
    last_time ← *current_time*
    last_temp ← *current_temp*
    Wait *next_interval* seconds
end while

Algorithm 2 Hourly Average Rate of Change Based Polling Algorithm

Initialize: *SENSOR_API_URL* as URL to fetch sensor data
Initialize: *THINGSBOARD_API_URL* as URL to post data to ThingsBoard
Initialize: *TARGET_DIFFERENCE* as desired temperature change threshold
Initialize: *HOURLY_RATE_OF_CHANGE* as array of average changes per hour
function GetSensorData
    Fetch data from *SENSOR_API_URL*
    Extract *timestamp* and *temperature* from the data
    return *timestamp*, *temperature*
end function
function PostToThingsBoard(payload)
    Set headers to {'Content-Type': 'application/json'}
    Post *payload* to *THINGSBOARD_API_URL* with headers
    return HTTP status code of the post request
end function
while true **do**
    current_time, *current_temp* ← GetSensorData
    current_hour ← Extract hour from *current_time*
    rate_of_change ← *HOURLY_RATE_OF_CHANGE*[*current_hour*]
    next_interval ← *TARGET_DIFFERENCE* / *rate_of_change*
    payload ← {'temperature': *current_temp*}
    PostToThingsBoard(payload)
    Wait *next_interval* seconds
end while