

Article

Deep-Sleep for Stateful IoT Edge Devices

Augusto Ciuffoletti Dipartimento di Informatica, Università di Pisa, I-56126 Pisa, Italy; augusto.ciuffoletti@unipi.it

Abstract: In an IoT (Internet of Things) system, the autonomy of battery-operated edge devices is of paramount importance. When such devices operate intermittently, reducing power consumption during standby improves such a characteristic. The *deep-sleep* operation mode obtains such a result: it keeps on power only the hardware needed to wake up the unit after a timeout or an external trigger. For this reason, deep sleep exhibits the issue of losing the working memory, which prevents its use with applications depending on long-lasting or *stateful* computations. A way to circumvent such an issue consists of saving a snapshot of the working memory on a remote repository. However, such a solution is not always convenient since it exhibits an energy footprint due to checkpoint transmission. This article analyzes the applicability of such a solution. Firstly, by comparing its energy footprint against keeping the working memory on power. The analysis follows a formal, technology-agnostic methodology based on a mathematical model for energy consumption. It yields a *discriminant inequality* identifying the use cases where remote checkpointing is of interest. Once justified the approach, the article proceeds by defining an architecture and a secure protocol for data transport and storage. Finally, the description of a prototype implementation provides concrete insights.

Keywords: internet of things; edge devices; duty-cycle; deep sleep; checkpointing; long-running computation; key-value database



Citation: Ciuffoletti, A. Deep-Sleep for Stateful IoT Edge Devices. *Information* **2022**, *13*, 156. <https://doi.org/10.3390/info13030156>

Academic Editor: Shingo Yamaguchi

Received: 1 February 2022

Accepted: 14 March 2022

Published: 17 March 2022

Publisher's Note: MDPI stays neutral with regard to jurisdictional claims in published maps and institutional affiliations.



Copyright: © 2022 by the author. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (<https://creativecommons.org/licenses/by/4.0/>).

1. Introduction

IoT (Internet of Things) is a technology with an impact on society witnessed by applications in traffic monitoring [1], e-health systems [2], public appliances control [3], and parking location assistance [4]. Several research projects demonstrate that IoT can improve the social inclusion of impaired people [5,6], and help the sustainable growth of developing countries [7].

From a designer's point of view, an IoT system is a hierarchical infrastructure processing a flow of data to and from autonomous *edge devices* interacting with the environment. Frequently, edge units cannot reach the electrical grid, so the designer applies energy harvesting and accumulators as power sources. In that scenario, using energy-saving techniques is mandatory to improve the autonomy of edge units.

One of such techniques exploits a peculiarity of sensors and actuators: their operation is often sporadic, triggered by external events, or scheduled periodically. When this is the case, the microcontroller driving the edge unit can save power, during the time between two activity periods, by powering only the hardware needed to resume the operation: manufacturers provide several power-management tools to this end.

In that direction, *deep sleep* is the ultimate energy-saving measure, since it keeps on power only the hardware needed for microcontroller wake-up: as a consequence, the residual current is in the range of μW and, in theory, a 3000 J charge (corresponding to a pack of three AAA rechargeable Ni-MH batteries) would last 3×10^9 seconds, one century. In practice, if the designer uses the deep-sleep mode only when the device is idle, energy consumption matches that required by its sporadic *payload* activity. In that framework, the term *duty-cycle* indicates the rate between the time during which the unit is functional

(i.e., not idle) and the length of the complete cycle, including the standby period. For instance, operating 1 min every hour corresponds to a duty cycle of 1.7%. In that case, using deep sleep, the accumulator lasts 60 times longer, or a solar panel is 60 times smaller. In a concrete case, a microcontroller that drains 100 mW from a 3000 J battery has an autonomy of eight hours; with the above duty cycle and deep sleep, the same battery lasts 20 days.

During deep sleep, the volatile working memory, which has a substantial impact on the power consumption of the micro-controller, is powered off and therefore loses its content. Therefore long-running, *stateful* applications that, by design, need to retain and update the working memory are at first sight incompatible with deep sleep since it erases the accumulated knowledge, the state of the stateful application. To address such an issue, MCU manufacturers provide light-sleep modes which prevent the loss of volatile memory at the cost of higher power consumption.

However, the designer has a way of using deep sleep duty cycling with stateful edge applications. It consists of storing the content of the volatile memory on a remote repository service during the deep sleep periods. The drawback of such a solution is evident since data transfer has a non-negligible energy footprint and suggests why the literature overlooks such a design direction. This paper aims at filling the gap by answering first a research question: “is there a point of balance between the power consumption of the volatile memory during standby and that for checkpoint transport?”.

To find an answer, we compare the energy consumption of the two alternative approaches. The expressions for the energy consumption use system parameters that characterize a use case. Therefore, the resulting inequality splits the space of the use-cases into two regions, one of which contains those for which remote storage is more convenient than light sleep.

Once demonstrated, with the help of a concrete use-case, that such a region is non-empty, the paper proceeds by defining an infrastructure that supports the storage service. The description covers the security aspects related to the transport and management of the volatile memory snapshots or *checkpoints*.

The final section introduces a complete and fully functional proof-of-concept implementation. The software is available on GitHub, and the hardware design uses widely available low-cost devices, which facilitates further investigations. The purpose of the prototype is to corroborate the claim that the provided architecture is a consistent solution for remote checkpointing.

2. Overview of Related Research

The subject of this paper is new in the literature about IoT systems. For this reason, this section is not a comparison with similar results but defines the relationships of the topics investigated in this paper with others present in the literature.

Duty cycling is a prominent feature of IoT systems, and therefore it is broadly studied [8]. An extensively investigated aspect addresses the functions that are necessarily kept on power while on standby. For instance, if a unit must be responsive to network requests during a standby period, one of such functions is the network interface [9]. A solution well-represented in the literature consists of introducing an additional receiver devoted to switching on the unit, called wake-up radio. Given the specific purpose, communication may adopt low-energy techniques. Kozłowski et al. [10] introduce a clock synchronization protocol to switch on the wake-up radio at defined times, introducing an instance of synchronized duty-cycling, which is another extensively studied topic [9].

Working memory is another function that may remain active during a deep-sleep period: this is the case when the application run by the edge unit depends on data collected during long periods, as in the case of long-lasting computations. These features require powering a substantial quantity of memory during the standby period. Studies aiming at lightweight time-series databases specifically designed for IoT systems [11], or edge units with continual learning capabilities [12] also address long-lasting computations in IoT. Recent commercial devices address such an issue as well: the MAX 78000 board [13,14],

based on the Cortex-M4 microcontroller, provides a 128KB low-power memory which keeps its content during standby periods.

Kazdaridis et al. [15] carry out an in-depth investigation about the duty-cycling technique by analyzing several alternative implementations, looking for the one with minimal power consumption. They propose an ad hoc board called *ICARUS* using a microcontroller with a low power consumption featuring an external real-time clock and a persistent memory implemented with a 32KB FRAM (Ferroelectric Random Access Memory) module.

IoT is inseparable from networking by definition, and IoT infrastructures consider that edge devices work in conjunction with remote services; for instance, they may perform data filtering and aggregation to reduce the amount of data produced by the system [16]. Such services run on other devices in the same infrastructure or the cloud [17]. Therefore the presence of remote services is inherent to IoT systems. However, the literature does not report about their utilization for duty cycling.

The present paper focuses on this topic, finding how to exploit networked resources for the persistent storage of edge working memory, thus allowing deep sleep with minimal use of local persistent memory.

Such a function is an instance of a generic checkpointing operation. Several articles consider checkpoint recording in IoT systems to cope with unanticipated events related to the power supply, such as the paper by Ghodsi et al. [18], which studies the optimization of checkpointing operations for intermittently powered edge units. Mirhoseini et al. apply the same approach to the peculiar case of long-running computations [19]. Gelenbe et al. introduce a cost function that takes into account the energy spent in recording and the computation time, showing how to find an optimum balance [20]. A more practical approach is in a paper by Jayakumar et al. [21] introducing a layered memory architecture using the FRAM technology.

The present paper is at the confluence of such diverse approaches since it explores remote checkpointing in deep-sleep duty cycling. Since the idea is new, we check its validity first, using a technology-agnostic methodology that ensures the future value of the results. Energy consumption is the parameter that guides the comparison between the alternative approaches.

3. Discriminating Favorable Use Cases

This section compares a *local* and a *remote* solution to power-saving. The former envisions deep sleep and remote checkpointing, while the latter uses light sleep periods keeping the local memory on power. Their energy footprints comparison allows discriminating the use-cases for which the remote approach is preferable.

The methodology to discriminate the use cases consists of characterizing each of them with a t -uple containing the relevant parameters of the edge unit hardware and software. Two models of the edge unit operation allow computing the energy consumption for the remote and the local solution starting from the use case t -uple. The comparison between the two provides the analytical response to the question.

The first step consists of defining the parameters in the t -uple. To this end, let us analyze the timing of edge device operation.

The generic edge unit operates according to a cycle, alternating a busy and a standby period controlled by a wake-up timer. However, any other random event generation process is suitable, provided that an average value for the period is defined. The duration of the cycle, or *period*, is Δt .

The criterion to discriminate between the two approaches is the energy consumption during an operation cycle. Assume that this latter consists of time intervals, each dedicated to a specific activity, characterized by a stationary power consumption related to the operation mode. Table 1 lists five kinds of activity, with associated parameters for the duration and power consumption. The subscripts of the parameters indicate the operation mode:

- *tr*: the radio is in transmission mode;
- *op*: normal operation, the whole device except the modem is powered;
- *ls*: light sleep, only the volatile working memory is powered;
- *ds*: deep sleep, only the wake-up circuit is powered.

Table 1. Notation of duration and power consumption for each kind of activity.

Activity	Duration	Power	Color
transmission	Δt_{tr}	P_{tr}	
payload	Δt_{pd}	P_{op}	
protocol	Δt_{pr}	P_{op}	
light-sleep	Δt_{ls}	P_{ls}	
deep-sleep	Δt_{ds}	P_{ds}	

The *t*-uple that defines a use case contains the durations and power consumptions listed in Table 1 and Δt_i . We assume the following relationship holds among power consumptions:

$$P_{tr} > P_{op} > P_{ls} \gg P_{ds} \tag{1}$$

The duration of a complete operation cycle (Δt) is an application requirement, and therefore it is identical for both the *remote* and *local* solutions. The same holds for the duration of the payload operation Δt_{pd} , since, in both cases, it refers to identical computations.

Figure 1 show the models of operation of the edge unit in the two cases as simplified, unscaled graphs of power consumption variation.

The area of each column in the graphs represents the energy spent in one of the activities in Table 1. As said, the values of Δt_{pd} and Δt are the same in the two figures since they depend on the application requirements and not on the duty-cycling solution.

The energy consumption for the local solution (in Figure 1a) is split into that consumed during normal operation and during light-sleep. The edge unit never enters transmission or deep-sleep modes. During the time interval Δt_{op} (in blue), the unit performs its task and then enters a light-sleep during the following Δt_{ls} (in green).

The energy consumption for the local solution E_{local} corresponds to the following expression:

$$E_{local} = \Delta t_{pd} P_{op} + (\Delta t - \Delta t_{pd}) P_{ls} \tag{2}$$

The operation cycle of the remote solution (see Figure 1b) starts by initializing the network interface and the checkpoint download protocol. The energy consumption in this phase is in light blue in the graph. Next, the unit starts retrieving the checkpoint, with a sequence of receive operations with occasional transmissions: the figure shows in red a request and a final acknowledgment, each contributing to the overall Δt_{tr} . Other protocol-related operations, such as data unmarshalling, follow the receive operation. The payload operation follows (in dark blue), which is the same carried out in the case of the local solution. Next, the edge unit prepares and sends the checkpoint, which entails networking activity that accounts for energy consumption, and finally enters the deep-sleep state. The equation for the energy consumption E_{remote} is the following:

$$E_{remote} = \Delta t_{pd} P_{op} + \Delta t_{pr} P_{op} + \Delta t_{tr} P_{tr} + (\Delta t - (\Delta t_{pr} + \Delta t_{tr} + \Delta t_{pd})) P_{ds} \tag{3}$$

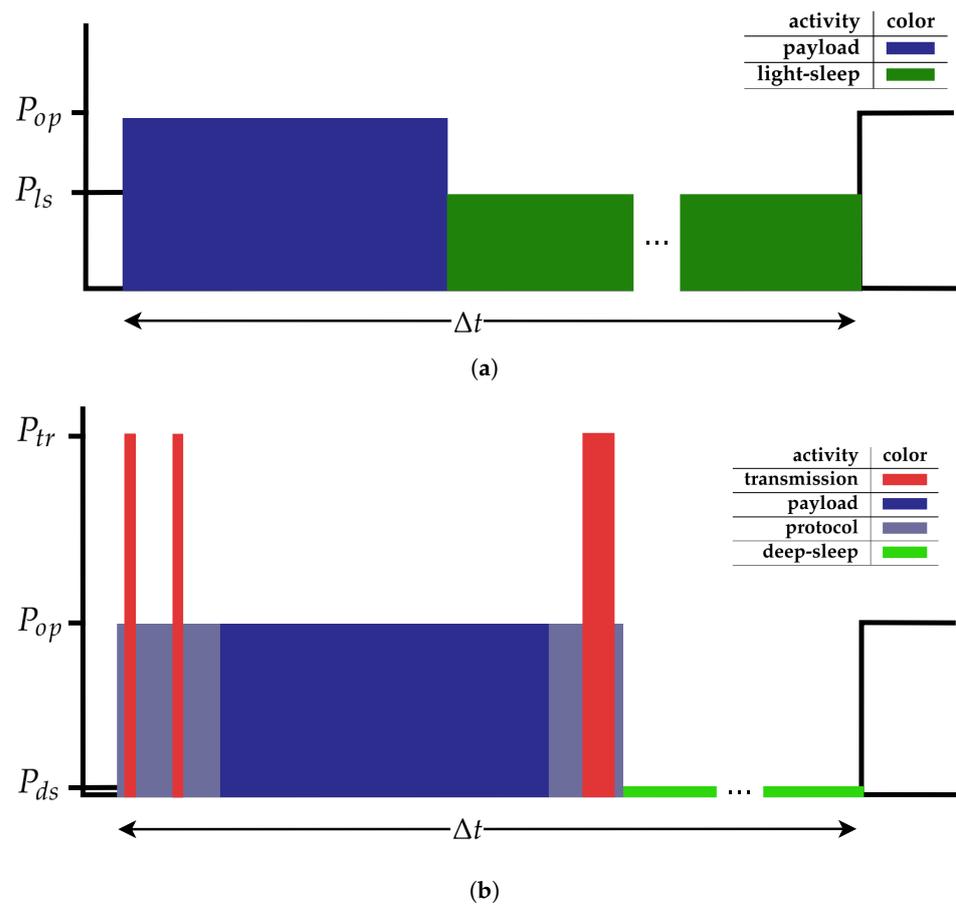


Figure 1. Outline of power consumption for the local and remote solutions. (a) Outline of power consumption for a local solution based on light sleep; (b) outline of power consumption for a remote solution based on deep sleep with remote checkpointing.

The inequality $E_{local} > E_{remote}$ holds when the remote solution is preferable to the local one. Replacing the two terms with those indicated in Equations (2) and (3), the inequality depends on the use-case parameters. With algebraic manipulation (see the appendix for the calculus), we obtain the *discriminant inequality*:

$$\Delta t > \Delta t_{pr} \frac{P_{op}}{P_{ls}} + \Delta t_{tr} \frac{P_{tr}}{P_{ls}} + \Delta t_{pd} \tag{4}$$

By replacing the parameters of a use-case in the inequality, we obtain a truth value that answers the question: “In the defined use-case, is it worth applying deep-sleep with remote checkpointing instead of light-sleep?”. The following section analyzes the discriminant to find conceptual guidelines for the applicability of the remote solution. Finally, we use it to find a concrete use case where the remote solution is preferable.

3.1. Discussion

The critical parameter in the discriminant is the operation cycle duration Δt , on the left of the inequality sign. This fact means that, in a given use-case described by the parameters on the right side, the remote solution is the most appropriate when the cycle is sufficiently long to balance the energy consumed by the memory during the stand-by period with that needed to transfer a checkpoint. The ideal candidate for a remote solution is a use-case with a long Δt .

The discriminant in equation (4) provides further insight into the region of the use-cases space for which a remote solution is preferable.

We notice that three parameters initially considered in the t -uple are not considered in the discriminant: $P_{ds}, \Delta t_{ds}, \Delta t_{ls}$. The absence of Δt_{ls} is justified since one of the duration parameters is redundant. The absence of the two parameters concerning the deep sleep period, P_{ds} and Δt_{ds} , depends on inequality (1), and is justified looking to the calculus in the appendix: the terms containing such parameters are negligible compared with the rest of the equation, and therefore they are eliminated to simplify the formula. This simplification gives a rationale to the common sense statement anticipated in Section 1: the energy consumed by an application that implements deep-sleep duty cycling approximately equals that of its sporadic operation periods. In conclusion, the t -uple defining a use case is:

$$[\Delta t, \Delta t_{pr}, \Delta t_{tr}, \Delta t_{pd}, P_{op}, P_{ls}, P_{tr}]$$

The term on the right of the discriminant has three addends, and each of them restricts the applicability region of the remote solution. The first one states that a communication protocol with low computational overhead (Δt_{pr}) extends such a region, a usual requirement for IoT applications.

Two addends inversely depend on the energy footprint of the light sleep modality P_{ls} . This parameter depends on the fraction of hardware powered in light-sleep mode, including the working memory. Assuming the memory that should be on power during light sleep corresponds to the checkpoint in the remote approach, the denominator is also proportional to the checkpoint size C_s . The constant rate P_b is the power required by one single bit in memory:

$$P_{ls} = P_b C_s \tag{5}$$

The following inequality is equivalent to the discriminant in Equation (4):

$$\Delta t > \Delta t_{pr} \frac{P_{op}}{P_b C_s} + \Delta t_{tr} \frac{P_{tr}}{P_b C_s} + \Delta t_{pd} \tag{6}$$

Such an expression reveals that use-cases making heavy use of persistent data are good candidates for a remote solution.

Since the duration of checkpoint transfer Δt_{tr} depends on its size, the numerator of the second addend depends on memory size too. In light of this, the role played by C_s looks controversial because it is both favorable and unfavorable for remote checkpointing. To resolve such an ambiguity, we replace the numerator $\Delta t_{tr} P_{tr}$ of the second term with the product between the checkpoint size C_s and the energy-per-bit rate E_b for the network:

$$\Delta t_{tr} P_{tr} = E_b C_s \tag{7}$$

Replacing such expression in the second term, a different formulation of the discriminant is:

$$\Delta t > \Delta t_{pr} \frac{P_{op}}{P_b C_s} + \frac{E_b}{P_b} + \Delta t_{pd} \tag{8}$$

The inequality confirms the role of the checkpoint size: the use of sizable checkpoints addresses a remote solution. In addition, the equation highlights the impact of the energy efficiency of the network technology (E_b): remote checkpointing is hardly applicable to scenarios adopting network technologies with a high E_b rate, such as LoRaWAN (long-range low-power networks).

This observation introduces an alternate definition for the t -uple defining a use case:

$$[\Delta t, C_s, \Delta t_{pr}, \Delta t_{pd}, P_{op}, P_b, E_b] \tag{9}$$

Equation (8) and the corresponding t -uple (9) explain the influence of the technology-dependent parameters P_b and E_b .

The last addend on the left side of the discriminant pertains to the payload computation associated with the application. As a consequence, remote checkpointing is a valuable option when the time taken by such operation is short compared with the cycle

time. This statement matches the intuitive observation made in the introduction that the remote approach is justified for applications with a low duty cycle.

The following section finds one use case for which remote checkpointing is more convenient. The exercise starts defining the application parameters and proceeds by characterizing the others.

3.1.1. Finding a Use Case

This section applies the concepts introduced in the previous section in the definition of a use case. The exercise uses Equation (8), using P_{ls} instead of the product $P_b C_s$.

The first step is the definition of the application function and scenario, which provides the values for the fundamental parameters of the target application, i.e., Δt and C_s . In the successive step, the use case includes a networking technology, which provides Δ_{pr} and E_b . Given the power consumption of the edge unit in normal operation P_{op} , Equation (8) is rewritten to return an alternate discriminant for P_{ls} :

$$P_{ls} > \frac{\Delta t_{pr} P_{op} - E_b C_s}{\Delta t - \Delta t_{pd}} \quad (10)$$

The inequality means that a remote solution is not of interest if power consumption in light-sleep mode is lower than the computed threshold.

The target application is a key-based door-lock. The users come in groups, and each individual has to authenticate. The scenario indicates an average interval of one hour between successive group checks, and the maximum number of granted authorization keys is 64. During operation, the device holds all of them in the working memory. Therefore, the value of Δt is 3600 s, and, assuming each record is 64 bytes long, checkpoint dimension C_s is 2 KB.

A limited budget justifies the utilization of a pre-existent WiFi infrastructure. The protocol management, which includes registering with the access point, takes an average of 10 s (Δt_{pr}) for each activity round. The network exhibits a high bitrate with a low energy-per-bit falling in the range of 10 nJ.

The operation includes human interaction for the presentation or creation of credentials and data management so that t_{pd} is estimated as 10 s.

The power consumption of the control board P_{op} is 150 mW.

In such a scenario, summarized in Table 2, the designer wants to know the threshold power consumption in light-sleep (P_{ls}) that would make preferable the remote checkpointing approach. The result, obtained by replacing the values of Table 2 in Equation (10), is 0.5 mW. It is six times lower than that documented for the ESP8266 microcontroller selected for cost and availability reasons (see Table 3). So the designer should either implement a remote solution, or propose a different microcontroller.

Table 2. Specifications for the use case. The threshold for local checkpointing of P_{ls} is of 0.5 mW. A higher value recommends the adoption of remote checkpointing.

Parameter	Value	Unit	Notes
Δt	3600	s	Cycle time
C_s	2×10^3	bytes	Checkpoint size
Δ_{pr}	10	s	Time for protocol management
E_b	10×10^{-9}	J	Networking E_b parameter
Δ_{pd}	10	s	Time for payload function
P_{op}	150×10^{-3}	W	Power in operation

Table 3. Power consumption in normal and power-saving modes [23] with 3.3 V power supply.

Mode Description	Current (mA)	Power (at 3.3 V)	Parameter Name
Modem transmission 802.11 g	140 mA	460 mW	P_{tx}
Normal operation	80 mA	260 mW	P_{op}
CPU suspended	0.9 mA	3 mW	P_{ls}
Timer only	0.02 mA	0.07 mW	P_{ds}

The valuable result here is that, in the specific use case, the remote solution exhibits a lighter energy footprint.

4. A Checkpoint Storage Service

The previous section justifies interest in remote checkpointing by proving that it is more convenient than local memory when system parameters fall in a defined region. This section outlines a remote checkpointing infrastructure consisting of a checkpoint storage server supporting many edge units.

The first step in this direction is to design the secure management of edge unit identifiers.

Burning such an identifier in the persistent *Flash* memory of the device is not a practical solution for the following reasons:

- It requires editing the edge device firmware, which is unreliable;
- The operator needs skills and tools to build and burn the firmware in the edge device;
- An error in the association (e.g., duplication) leads to coordination problems;
- The operator has access to a sensible piece of information, thus degrading service security.

Using a factory-configured identifier in analogy to a MAC address exhibits similar problems. The following use case describes the limits of such a solution.

An e-health system uses microcontroller-driven stations to monitor patient's health parameters. If the operator in charge of installing a new station knows the user identifier, such data grants access to the patient's data in the checkpoint server, thus creating a security issue. The encryption of stored data does not solve the problem since the decryption key should be available in the edge device memory, reintroducing the problems already seen above.

A better solution consists of a protocol that dynamically associates an identifier to an edge unit, possibly assisted by an operator but without exposing the identifier associated with the edge unit.

4.1. Where All Begins: Identity Creation

A fundamental requirement for an identifier is uniqueness. The reasons seen exclude a static, administration-driven configuration, but the server is another component that can guarantee uniqueness with a dynamic identifier allocation. A protocol that associates a lightweight operation with reasonable and tunable security is the following (see Algorithm 1, which also reports features discussed in the following sections): upon reset, the edge unit generates a random identifier and submits it to the server, which checks its uniqueness and sends back an acknowledgment (line 4–6 in Algorithm 1). Notably, the server must ensure that the check-and-send operation runs as an indivisible transaction to avoid that two simultaneous submissions with identical identifiers both receive a positive acknowledgment.

When the edge unit receives the acknowledgment, it records the checked identifier in a persistent memory of limited size (line 14). At the end of each operation cycle, the edge unit records the working memory as a checkpoint in the remote storage, using the identifier as an index (lines 17–18). At the beginning of the following operation cycle, the edge unit uses the same identifier to download the remote checkpoint and initialize the working memory (line 12).

The following section discusses security problems related to this protocol and its implementation as a key-value database.

Algorithm 1 Algorithm run by the edge unit.

```

1: variable Idx: integer
2: joinAP(pass, SSID)
3: if system.getFlag(reset) then                                ▷ unit has been reset
4:   repeat
5:     Idx := random()
6:   until http.head("https://storage.local/"+Idx) != 200        ▷ check Idx not present
7:   operatorSend(Idx)                                          ▷ ask operator's ack
8:   while http.head("https://storage.local/"+Idx) != 200 do  ▷ check Idx is present
9:     delay(5)
10: else                                                         ▷ unit wakes up after deep sleep
11:   Idx := system.readRTCreg()                                ▷ unit reads id from non-volatile register
12: http.get("https://storage.local/"+Idx, response)
13: nextId := response.newId                                    ▷ acquires next Id from the server
14: system.writeRTCreg(response.nextId)                        ▷ store next id in non-volatile register
15: unpackState(response.checkpoint)
16: businessLogic()
17: request.checkpoint := packState()
18: http.post("https://storage.local/"+nextId, request)
19: system.deepSleep()

```

4.2. Security Issues

The above protocol for identity creation is exposed to an attack consisting of flooding the server with malicious submissions, thus saturating the database capacity with resulting service unavailability.

The presence of an authorized operator during edge device initialization may avoid this sort of problem: the operator acts as an intermediary using an authenticated mobile phone application (see Algorithm 2), which authorizes the delivery of a new identifier, thus preventing identifiers exhaustion. The following three-party protocol implements such a mechanism:

- The edge device generates and submits a new identifier (line 4–6 in Algorithm 1);
- The server checks uniqueness and returns an acknowledgment without committing the reservation;
- Upon receiving the acknowledgment, the edge device asks the operator to authorize the reservation (line 7 of Algorithm 1). Next, it start checking the creation of the new identifier on the server, with repeated failures, since the server did not commit the reservation (line 8–9 of Algorithm 1);
- The operator uses the authenticated connection with the server to authorize the reservation (line 4 of Algorithm 2);
- The server commits the reservation, and the edge device succeeds in downloading the initial checkpoint which possibly contain factory configurations (line 12 of Algorithm 1);
- In the marginal case another edge unit simultaneously obtains the same identifier the server returns an error to the operator. Error detection induces a reboot, and the process repeats (line 8 of Algorithm 2).

Algorithm 2 The operator's mobile phone application.

```

1: loop
2:   Idx := BLE.poll()           ▷ Poll bluetooth input to receive id from unit
3:   request.body := factoryCfg   ▷ Initial checkpoint is factory configuration
4:   if http.put("https://storage.local/" + Idx, request) == 200 then
5:     system.display("Setup successful")
6:     system.exit()
7:   else
8:     system.display("Setup collision. Please reset the device")

```

With this protocol in place, an intruder cannot reserve an identifier, but more sophisticated attacks are still possible.

If the networking infrastructure exposes the packet content, a malicious third-party observer may discover the identifier associated with an edge unit. For this reason, the designer should consider the adoption of protocols with payload encryption.

The unlimited access to the checkpoints granted to the database administrator is also a potential security threat: a misbehaving administrator can compromise system integrity and leak sensitive data. The counter-measure for this kind of attack consists of encrypting each record on the source using an encryption key randomly generated by the edge unit and stored in a persistent register together with the identifier. Such a measure ensures that only that same edge device has access to the checkpoint, enforcing strong security. However, any form of online maintenance becomes impossible.

The checkpointing service is exposed to a brute force attack consisting of a massive trial of identifiers, looking for one already assigned to an edge unit and thus stealing its identity. This attack is successful in the long run since the server gives a transparent positive response to a request carrying a valid identifier, also exposing the checkpoint content.

The distribution of the time required by such an attack depends on the cardinality of the identifiers domain. Such cardinality depends on the bit-length of an identifier: with an n -bits identifier, a single trial has a probability of success of 2^{-n} . If at a given time m identifiers are in use, the intruder has a probability of $m 2^{-n}$ of guessing one. For instance, with a 32-bit identifier and a maximum of 1000 edge units in the system, the probability of guessing right is 2^{-22} , less than one over one million. However, after 5000 trials, the probability of success is significant, around 0.001. To contrast such attacks, the designer may delay the response, block suspected source addresses, and use longer identifiers.

Unauthorized access to the server is another security issue. In that case, the target of the intruder can be either the edge units identifiers or the checkpoints themselves. A measure that is specific for the remote checkpointing scenario is the *identifier swap*, which consists of changing the identifier associated with an edge device at each activity cycle. The server delivers the new identifier to the edge unit during checkpoint download (line 12–13 in Algorithm 1). The edge unit stores it in the local persistent register (line 14) and uses it during the next upload (line 18). The intruder that finds an identifier has access to a checkpoint for a limited time. After that time, the identifier is deallocated and it becomes useless.

As seen in the case of the disloyal database administrator, the edge unit may address content encryption to protect the checkpoint content. When considered for outbound protection, such a measure is similar in concept to the identifier-based access to the checkpoint storage. However, its effectiveness is limited since, once the intruder has an encrypted checkpoint available, the trial of millions of encryption keys becomes viable. Therefore it is preferable to extend the domain of the identifiers instead of encrypting the checkpoints.

To summarize, the security of the remote server relies on four measures:

- A networking infrastructure that prevents packet content exposure using encrypted communication;
- A wide range of identifiers to make identifier scanning unsuccessful with high probability;

- An identifier swap mechanism that limits the damage of the successful guess of an identifier;
- Protection of the physical device hosting the checkpoint database.

4.3. Server and Network Load

For the reasons explained in the previous section, the number of allocated identifiers m is small compared with the namespace dimension. Therefore, given that the server implements the database as a sparse array, the required capacity of the database storage is the product $m C_s$, where C_s is the checkpoint size.

The network infrastructure transports the checkpoint to and from the server. The data transfer bandwidth Bw_{data} is computed using the values in Table 1. Since each edge unit downloads and uploads a checkpoint of size C_s during an operational cycle lasting Δt , the average bandwidth for a system of maximum size m is:

$$\overline{Bw_{data}} = \frac{2 m C_s}{\Delta t} (\text{Byte} \times \text{s}^{-1}) \quad (11)$$

It is an average value, while the maximum load depends on the use case and the operating conditions.

Considering the use-case defined in Section 3.1.1, the maximum number of records is 64, each 2 KBytes long. The resulting database size is 128 KBytes. The average networking overhead depends on the average duration of the operational cycle, the checkpoint size, and the number of units in the system: Equation (11) returns a value of 437 Bytes/s.

5. Prototype Implementation

The purpose of this section is threefold:

- To provide evidence for the applicability of the abstract design;
- To demonstrate the availability of the parameters in the t -uple describing the use case;
- To give sufficient detail to reproduce the prototype for further study and improvement.

The prototype follows the guidelines given for the use case defined in Section 3.1.1. Its design targets widely available technologies and materials to simplify its reproduction [22]. It is not intended to show an optimal implementation.

5.1. Materials and Methods

The first step in the prototype design is the definition of the link-layer technology. We opt for the WiFi protocol because it is a low-cost and low-impact solution that may be of practical interest. The E_b rate meets the guidelines in Section 3.

A single-board computer suitable as the checkpoint server is the Raspberry Pi: the prototype uses a Version 3 Model B device, equipped with 1 GB RAM, supporting WiFi and Ethernet networking. The Operating System is a Raspian v10 Lite (codename *Buster*) installed onto a 4GB micro SD card. Suitably configured, a Raspberry Pi can implement a WiFi Access Point of limited capacity.

The microcontroller for the edge device is a Wemos D1 R2 board, mounting an ESP8266 12F module and a few additional hardware components for power supply and development. The microcontroller provides 80KB of volatile memory (SRAM), 1MB of flash memory for programs, and a WiFi interface. During deep sleep, the power management of the microcontroller feeds only a timer and a limited quantity of memory (512 Bytes).

The ESP8266 provides two other power-saving modes besides deep sleep. One, called *modem sleep* in manufacturer's terminology, switches off the WiFi modem. The other, called *light sleep*, excludes most of the microcontroller hardware modules but keeps the SRAM powered. This latter corresponds to the light sleep mode discussed in Section 3.

The current drains documented by the manufacturer's reference manual [23] are summarized in Table 3. Consider the nominal supply voltage of 3.3 V to convert them into power consumption. They are typical values that depend on operating conditions and refer

to the ESP8266EX module alone. Other onboard hardware components contribute to power consumption.

The network layer is an intranet managed by a DHCP server implemented by the Raspberry Pi on the WiFi interface.

5.2. Identifier Management Protocol

The TLS encrypted HTTP protocol implements the application-level communication between the edge units and the server. The server implements a REST interface that provides the checkpoint storage functionalities, as summarized in Table 4. The operation associated with HTTP verbs are shown in the Algorithm 3.

Algorithm 3 HTTP verbs processing on the checkpoint repository service.

```

1: procedure HEAD(path, request)
2:   key := path
3:   if redis.exists(key) then
4:     response.code := 200
5:   else
6:     response.code := 404
7:   return response
8: procedure GET(path, request)
9:   key := path
10:  response.checkpoint := redis.get(key)
11:  redis.delete(key)
12:  repeat
13:    newKey := random()
14:  until ! redis.set(newKey, empty, notExists)
15:  response.newKey := newKey
16:  response.code := 200
17:  return response
18: procedure POST(path, request)
19:  key := path
20:  if redis.exists(key) then
21:    redis.set(key,request.checkpoint)
22:    response.code := 200
23:  else
24:    response.code := 404
25:  return response
26: procedure PUT(path, request)
27:  key := path
28:  if redis.set(key, empty, notExists) then
29:    response.code := 200
30:  else
31:    response.code := 404
32:  return response

```

Table 4. Functions associated with REST verbs. Line number refer to Algorithm 3.

Verb	Endpoint	Function	Line in Algorithm 3
HEAD	Id	Check if <i>Id</i> is taken	1
GET	Id	Download checkpoint	8
POST	Id	Upload checkpoint	22
PUT	Id	Cscreate <i>Id</i>	30

Figure A1 in the Appendix B is an alternate representation of Algorithm 1 describing the Wemos edge unit operation.

The first part of the flow chart, in light gray, implements the identifier association, which occurs when an operator installs a new device. Starting from the *Power-up* state, the edge unit generates a random identifier (*Idx*). Next, it submits a HEAD request to the server using *Idx* as the URL endpoint. If the identifier is unused, which is most likely to occur, the server responds with a 404 response (line 6 in Algorithm 3). The edge unit proceeds by asking the operator to authorize the creation of the new identifier. The transaction uses a Bluetooth connection between an app on the operator's smartphone and the edge device (line 2 in Algorithm 2). In the unlikely case the identifier is already assigned to another edge device, the unit receives a 200 response to the HEAD request (line 4 in Algorithm 3), generates a new random identifier, and sends another HEAD request.

After asking the operator to authorize the identifier reservation, the unit starts looping on HEAD requests waiting for a 200 reply. Meanwhile, the smartphone app forwards a certified authorization request to the server as a PUT request (line 4 in Algorithm 2). If there are no colliding requests, the server allocates a database entry to the identifier (line 28–29 in Algorithm 3), thus completing the reservation. Otherwise, if the server receives two simultaneous requests for the same identifier, it returns a negative response to one of them (line 31 in Algorithm 3). This event determines the re-initialization of the whole process (line 8 in Algorithm 2).

When the edge unit receives a 200 response to its HEAD request, it enters the routine operation (light green in the flow chart) by issuing a GET request. The response contains the initial checkpoint for the edge unit (line 10 in Algorithm 3) together with another valid identifier (lines 12–15 in Algorithm 3), thus implementing the identifier swap function.

After executing the payload task, the unit issues a POST request to store a checkpoint associated with the new identifier. Before entering deep sleep, the edge unit sets up a wake-up timeout and records the identifier in the persistent storage.

At the end of the deep sleep period, the edge unit enters the flow chart in the *wake up* state. Next, the edge unit recovers from a persistent register the new identifier and downloads the last checkpoint. The unit operation proceeds with the GET request, the payload operation, and the final PUT.

Figure 2 gives a comprehensive example of the activity of the three components: server, edge unit, and the operator's mobile app. The colors associate the edge unit operation with the flowchart in Figure A1. In the example shown, the edge unit starts by guessing an identifier that is already associated with another edge unit and receives a 200 response. This event triggers a new guess. If the second guess is successful, the response has code 404. Then the unit proceeds by asking the operator to authorize the reservation of a new identifier. Next, the operator device issues the PUT request to the remote service to obtain the reservation. The remote service responds by creating the requested identifier with an atomic transaction. Concurrently, the unit sends HEAD requests to the remote service to check if the *Id* is in the database. When it receives a 200 response, it enters the routine operation (green highlight): first, it downloads the content of the checkpoint with a GET, possibly containing configuration data, and the next identifier to use. After completing the payload task, the unit issues a POST request containing the new checkpoint using the identifier received with the GET. Then it records that same identifier (*Id3* in the example) in the rewake memory and enters the deep-sleep mode. When the wake-up timer expires, the unit finds on the remote storage, associated with the identifier found in the rewake memory, the recorded checkpoint together with a new identifier.

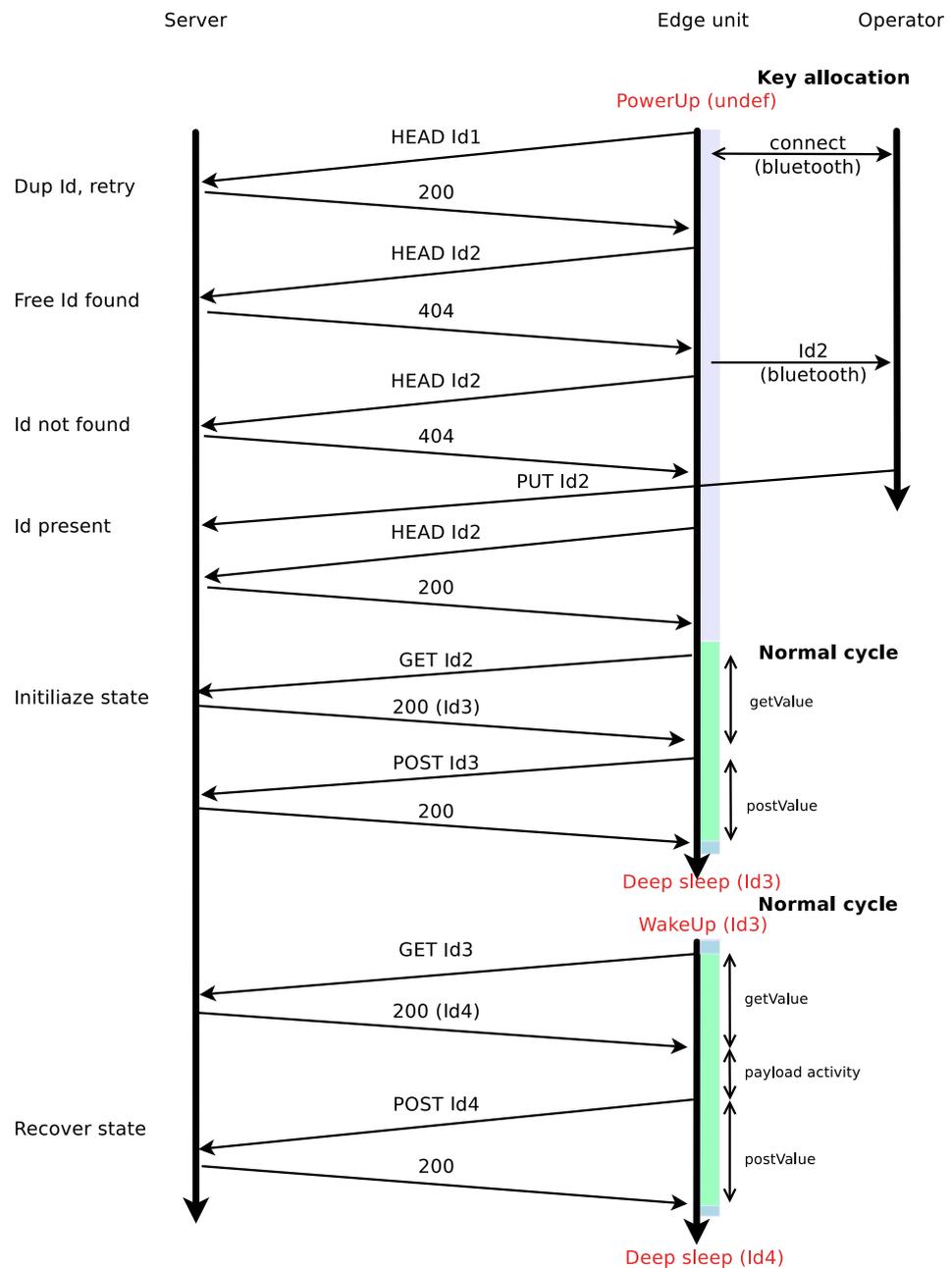


Figure 2. Example of a protocol run showing the interplay of the three roles. The first *Id* guess of the edge unit fails because the unit indicates an *Id* already assigned to another unit. The color codes on the edge unit timeline are the same as in Figure A1.

5.3. Server Implementation

The server, hosted by the Raspberry PI, is based on a Redis key-value database. This application implements an “in-store” key-value database service which ensures a lightweight operation. However, it requires that the device memory is sufficient for the task. In our prototype, after launching the server, the available memory exceeds 400 MB and overkills the memory requirements of the reference use case, computed in Section 4.3. In our perspective, the adoption of the Raspberry simplifies the implementation of alternative protocols and algorithms.

In the same perspective of easy prototyping, the software is modularized into three dockers following a standard schema: an HTTP proxy (Nginx), a WSGI server (Flask), and the Redis server. The HTTP proxy exposes a Secure HTTP port and routes the incoming requests to the WSGI server. This latter makes access to the Redis server. We

performed several tests to ensure that the Docker-based implementation is compatible with Raspberry capabilities.

The operation of the WSGI server, as outlined in the pseudocode in Algorithm 3, consists of translating REST requests into Redis API calls as follows:

- HEAD: performs a *get* using the *Id* as the Redis key and returns a status 200 if the key is present, and status 404 otherwise;
- GET: performs a *get* call on the Redis server using the *Id* in the URL endpoint as the Redis key. This function has the effect of downloading the requested checkpoint and removing it from the database. A new *Id* is generated (for identifier swapping), and returned to the requester in the same response;
- POST: performs a *set* call on the Redis database using the *Id* in the URL endpoint as the Redis key;
- PUT: performs a *set* call on the Redis database using the *Id* in the URL endpoint as the Redis key. The *set* call contains a parameter ensuring that the operation is successful only if the key is not present in the database. According to the specifications, the whole operation is atomic.

The source code for the server and the sketch for the edge unit are in distinct GitHub repositories, respectively, at <https://github.com/augustociuffoletti/kviot> (accessed on 15 December 2021) and https://github.com/augustociuffoletti/iottemplate_esp8266 (accessed on 15 December 2021). The Android App for the operator is written for AppInventor and is reachable through the above repositories.

5.4. Experimental Results

One reason for implementing a prototype was to verify the completeness and consistency of system specifications. To this purpose, the prototype was extensively used during many test sessions, also with high loads and injecting unlikely situations. For instance, one of the experiments overloaded the server with the equivalent of 100 edge units running with a Δt of 20 s. Such a load is 50 times higher than that required by the reference use case (namely 64 units with an activity period of 600 s). The server operation proved robust, with no significant degradation or inconsistency in overload conditions.

The implementation of a prototype also allowed us to verify the energy model described in Section 3. The model, synthesized in Figure 1a, hides the fine-grain details of the variations of energy consumption: the question is if the model preserves the relevant information.

Figure 3 is a fine grain display of current consumption during an activity period with a 2 KBytes checkpoint. The simple measurement circuit consists of a 1 Ohm shunt, visible in Figure 4b. In that way, one mV on display corresponds to a current of one mA. According to that, a division on the vertical scale corresponds to 25 mA. The horizontal division is 1s so that the display covers a 12 s period.

During the standby period, the current is significantly lower than during activity. In operation, the average current is around 55 mA. Evident spikes reach 180mA and are related to the WiFi modem activity. In the magnified view on the bottom of Figure 3 we observe a significant sequence of such spikes with a resolution of 2 ms per horizontal division. With such resolution, they appear as periods of less than 1ms with a current of 140 mA.

The blue trace on top of the screen is a marker signal: it goes down at the end of the WiFi join, goes up when the unit delivers the GET request and down when it receives a response, and then up again when it sends the POST, and down when it terminates the activity cycle. Using that trace as a reference, we see how the WiFi join operation is the most time-consuming (4.5 s). The GET and POST operations take 1.4 s each.

Comparing this image with the outline in Figure 1b we find the same elements, with similar relationships: outside the operation period, the power consumption is negligible (P_{ds}); during the operational cycle, the power consumption is steady (P_{op}), with occasional peaks corresponding to modem operation (P_{tr}). In the outline shown in Figure 1b such

peaks are localized; in reality, their distribution in time depends on the network driver design and technology. However, from the model perspective, their fine-grain distribution is immaterial: what is relevant is the total time spent in transmission.

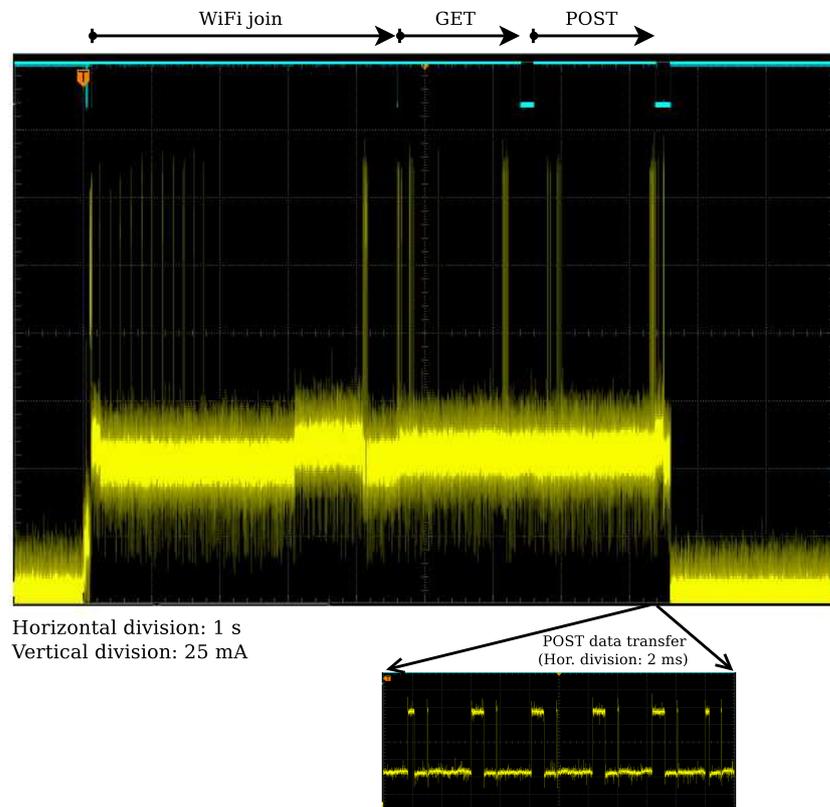


Figure 3. Voltage measured on the shunt resistor (1 Ohm) on the oscilloscope. The upper blue trace shows transition from the WiFi join operation (falling edge), to the HTTP GET (raising edge) and POST (falling edge).



Figure 4. (a) The two prototype components: an edge unit (left) and the checkpoint storage server (right); (b) current measurement on the edge unit. The channel 1 of the oscilloscope measures the voltage on a 1 Ohm shunt resistor in series with the MCU. Channel 2 visualizes marker signals injected in the code.

It is thus possible to derive from the trace in Figure 3 the parameters needed to compute the right-side expression in Equation (4). In addition, the dimension of the checkpoint is the same used in the use-case described in Section 3.1.1, 2 KBytes, and power supply is 3.3 V. Table 5 summarizes the resulting values.

Table 5. Parameters derived from Figure 3 except P_{ls} , extracted from manufacturer data-sheet.

Parameter	Value	Unit
P_{op}	200×10^{-3}	W
P_{ls}	900×10^{-6}	W
Δt_{pr}	8	s
Δt_{pd}	200×10^{-3}	s
P_{tr}	400×10^{-3}	W
Δt_{tr}	60×10^{-3}	s

Filling Equation (4) with the values in Table 5 we observe that the first addend, the one representing the impact of the network protocol, contributes for the 99% to the final value of 1800s, corresponding to 30 minutes. Instead, the addend that depends on the quantity of exchanged data has a limited impact. Thus, the designer may expect similar results with more extensive checkpoints. Furthermore, the energy spent for payload processing is not very significant: the threshold would not change much if the processing takes seconds instead of milliseconds.

In a long-term experiment with the same edge application, the power for the edge device came from an accumulator with a nominal capacity of 7×10^3 J. The test lasted more than 5000 operation cycles before the battery was exhausted. Such a result matches with the estimate of 4200 operation cycles, which is obtained from Equation (3) by substitution of the values in Table 5. With a Δ_t of 1800s indicated as the threshold, such a power source (which is equivalent to three low-cost AAA Ni-MH accumulators) provides an autonomy of three months. A compact Li-Ion battery, as the one shown in Figure 4a, may have more than five times that capacity, with an expected autonomy of more than one year.

In summary, the implementation of the prototype demonstrates that the overall architecture is sound and coherent with current technology and that the abstract model is adherent to the observed operation. This latter feature proves that the model may assist the designer in improving a design.

For instance, there is a clear indication that the energy footprint is mainly due to WiFi networking. However, this same technology makes negligible the footprint for data transfer. Such an indication suggests technologies with a comparable bitrate and faster management to the designer aiming at a lighter energy footprint.

6. Conclusions

This paper proves that remote checkpointing may enhance the energy efficiency of stateful edge applications. This conclusion is not immediate since remote storage operations entail energy consumption against energy-saving intents. However, it is possible to balance the energy spent for checkpoint transport by taking advantage of the drastic energy saving obtained with deep sleep during standby periods.

The interest in such a solution depends on application features that this paper defines in the first place. In essence, remote checkpointing targets stateful applications with a sufficiently long period and a state of significant size.

The region of interest starts from applications with moderate requirements. The paper shows that a few kilobytes of memory and a period of one hour is enough to compensate remote checkpointing costs induced by a WiFi network. However, the applicability range extends to use cases addressing different and possibly ad hoc hardware. For instance, to manage continual learning applications where the size of persistent memory is an expensive and limiting factor. From this perspective, the remote checkpointing approach makes accessible a class of demanding applications to low-cost hardware, as shown by the prototype implementation.

Once demonstrated that the approach is of interest, we provide a conceptual implementation describing the architecture and the communication protocol to implement remote checkpointing and an open source prototype.

The topic of this paper is stateful computation under energy constraints. Such framework includes applications that offer adaptive operation, possibly in conjunction with Artificial Intelligence techniques, and are the IoT technology cutting edge. The cost of low power memories or the need to branch the unit to the power grid limits this range of applications. Using remote checkpointing such applications become compatible with available low-cost hardware.

In the future, we may expect an improvement of all the figures considered in this paper. As long as the assumptions on system operations hold, the analytic criteria used to discriminate favorable use cases will remain valid, regardless of the quantitative improvements in communication technology, memory hardware, and processing capacity.

Finally, the approach followed in this investigation may be used as a blueprint to explore other innovative software architecture.

One research direction envisions the provision of multiple checkpoints that depend on environmental conditions: for instance, one during the day, another during the night. When the operation of the edge unit follows different schemas depending on external conditions, a device may check such external conditions before downloading the most suitable checkpoint. This technique would reduce the amount of onboard memory of learning devices.

The dynamic reconfiguration of edge devices is another conceivable evolution of remote checkpointing. It is similar to the On-The-Air (OTA) configuration technique but less intrusive. It consists of updating the device checkpoints present on the remote repository according to the changes in a dynamic scenario, as in the case of fault-tolerant applications. As discussed in the paper, the designer should place extra effort on security. The advantage would be avoiding the need for operator intervention, which would be beneficial in the case of hard-to-reach devices.

Funding: This research received no external funding.

Conflicts of Interest: The author declares no conflict of interest.

Appendix A. Detailed Calculus for the Discriminant Inequality

The following calculus shows how inequality (4) descends from $E_{local} > E_{remote}$ assuming $P_{tr} > P_{op} > P_{ls} \gg P_{ds}$.

$$\begin{aligned}
 E_{local} &> E_{remote} \\
 \Delta t_{pd} P_{op} + (\Delta t - \Delta t_{pd}) P_{ls} &> \Delta t_{pd} P_{op} + \Delta t_{pr} P_{op} + \Delta t_{tr} P_{tr} + (\Delta t - (\Delta t_{pr} + \Delta t_{tr} + \Delta t_{pd})) P_{ds} \\
 (\Delta t - \Delta t_{pd}) P_{ls} &> \Delta t_{pr} P_{op} + \Delta t_{tr} P_{tr} + (\Delta t - (\Delta t_{pr} + \Delta t_{tr} + \Delta t_{pd})) P_{ds} \\
 (\Delta t - \Delta t_{pd}) (P_{ls} - P_{ds}) &> \Delta t_{pr} P_{op} + \Delta t_{tr} P_{tr} - (\Delta t_{pr} + \Delta t_{tr}) P_{ds} \\
 (\Delta t - \Delta t_{pd}) (P_{ls} - P_{ds}) &> \Delta t_{pr} (P_{op} - P_{ds}) + \Delta t_{tr} (P_{tr} - P_{ds}) \\
 (\Delta t - \Delta t_{pd}) P_{ls} \left(1 - \frac{P_{ds}}{P_{ls}}\right) &> \Delta t_{pr} P_{op} \left(1 - \frac{P_{ds}}{P_{op}}\right) + \Delta t_{tr} P_{tr} \left(1 - \frac{P_{ds}}{P_{tr}}\right)
 \end{aligned}$$

From the assumption:

- $\left(1 - \frac{P_{ds}}{P_{ls}}\right) \approx 1$ since $P_{ls} \gg P_{ds}$
- $\left(1 - \frac{P_{ds}}{P_{op}}\right) \approx 1$ since $P_{op} \gg P_{ds}$
- $\left(1 - \frac{P_{ds}}{P_{tr}}\right) \approx 1$ since $P_{tr} \gg P_{ds}$

Then:

$$(\Delta t - \Delta t_{pd}) P_{ls} > \Delta t_{pr} P_{op} + \Delta t_{tr} P_{tr}$$

$$\Delta t > \Delta t_{pr} \frac{P_{op}}{P_{ls}} + \Delta t_{tr} \frac{P_{tr}}{P_{ls}} + \Delta t_{pd}$$

Appendix B. Flow Chart of Edge Unit Operation

A flow chart of a single activity cycle of a sensor unit. The operation starts from the *power-up* entry point, when the edge unit does not have an identifier and needs to create one, or from the *wake-up* entry-point when the edge unit emerges from a deep-sleep period.

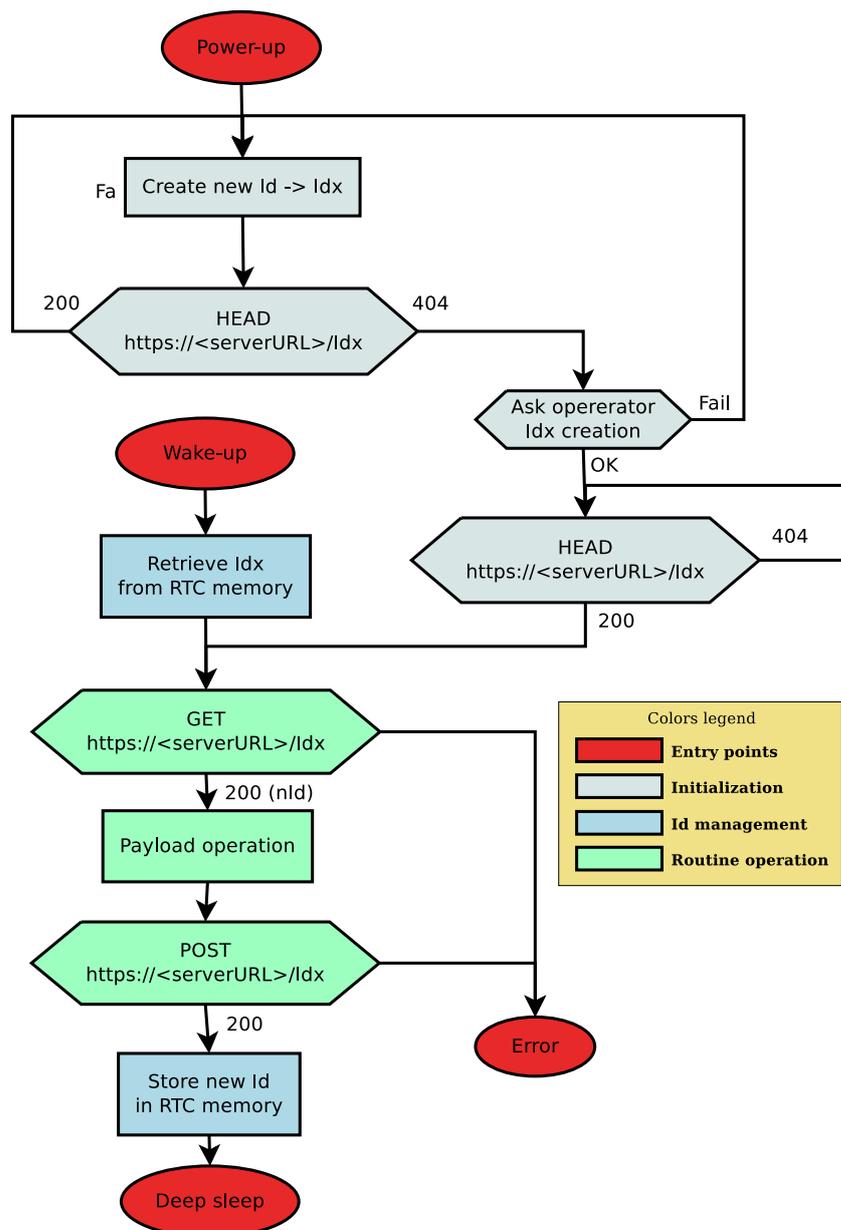


Figure A1. Flow chart of edge unit operation.

References

1. Periša, M.; Cvitić, I.; Peraković, D.; Husnjak, S. Beacon technology for real-time informing the traffic network users about the environment. *Transport* **2019**, *34*, 373–382. [CrossRef]
2. Ndiaye, M.; Oyewobi, S.S.; Abu-Mahfouz, A.M.; Hancke, G.P.; Kurien, A.M.; Djouani, K. IoT in the Wake of COVID-19: A Survey on Contributions, Challenges and Evolution. *IEEE Access* **2020**, *8*, 186821–186839. [CrossRef] [PubMed]
3. Hoon Kim, T.; Ramos, C.; Mohammed, S. Smart City and IoT. *Future Gener. Comput. Syst.* **2017**, *76*, 159–162. [CrossRef]
4. Lookmuang, R.; Nambut, K.; Usanavasin, S. Smart parking using IoT technology. In Proceedings of the 2018 5th International Conference on Business and Industrial Research (ICBIR), Bangkok, Thailand, 17–18 May 2018; pp. 1–6. [CrossRef]
5. Garcia, C.; Fernandes, P.; Davet, P.; Lopes, J.a.L.; Yamin, A.; Geyer, C. A Proposal Based on IoT for Social Inclusion of People with Visual Impairment. In Proceedings of the 23rd Brazillian Symposium on Multimedia and the Web, Gramado, RS, Brazil, 17–20 October 2017; Association for Computing Machinery: New York, NY, USA, 2017; pp. 489–495. [CrossRef]
6. Peraković, D.; Periša, M.; Remenar, V. Model of guidance for visually impaired persons in the traffic network. *Transp. Res. Part F Traffic Psychol. Behav.* **2015**, *31*, 1–11. [CrossRef]
7. Ugwuanyi, S.; Paul, G.; Irvine, J. Survey of IoT for Developing Countries: Performance Analysis of LoRaWAN and Cellular NB-IoT Networks. *Electronics* **2021**, *10*, 2224. [CrossRef]
8. Haimour, J.; Abu-Sharkh, O. Energy Efficient Sleep/Wake-up Techniques for IOT: A survey. In Proceedings of the 2019 IEEE Jordan International Joint Conference on Electrical Engineering and Information Technology (JEEIT), Amman, Jordan, 9–11 April 2019; pp. 478–484. [CrossRef]
9. Carrano, R.C.; Passos, D.; Magalhaes, L.C.S.; Albuquerque, C.V.N. Survey and Taxonomy of Duty Cycling Mechanisms in Wireless Sensor Networks. *IEEE Commun. Surv. Tutor.* **2014**, *16*, 181–194. [CrossRef]
10. Kozłowski, A.; Sosnowski, J. Energy efficiency trade-off between duty-cycling and wake-up radio techniques in IoT networks. *Wirel. Pers. Commun.* **2019**, *107*, 1951–1971. [CrossRef]
11. Lan, L.; Shi, R.; Wang, B.; Zhang, L.; Shi, J. A Lightweight Time Series Main-Memory Database for IoT Real-Time Services. In *Internet of Vehicles. Technologies and Services Toward Smart Cities, Proceedings of the 6th International Conference, IOV 2019, Kaohsiung, Taiwan, 18–21 November 2019*; Hsu, C.H., Kallel, S., Lan, K.C., Zheng, Z., Eds.; Springer International Publishing: Cham, Switzerland, 2020; pp. 220–236.
12. Ravaglia, L.; Rusci, M.; Nadalini, D.; Capotondi, A.; Conti, F.; Benini, L. A TinyML Platform for On-Device Continual Learning with Quantized Latent Replays. *IEEE J. Emerg. Sel. Top. Circuits Syst.* **2021**, *11*, 789–802. [CrossRef]
13. Maxim Integrated Products. MAX78000 User Guide. Available online: <https://pdfserv.maximintegrated.com/en/an/ug7456.pdf> (accessed on 15 December 2021).
14. Giordano, M.; Magno, M. A Battery-Free Long-Range Wireless Smart Camera for Face Recognition. In Proceedings of the 19th ACM Conference on Embedded Networked Sensor Systems, Coimbra, Portugal, 15–17 November 2021; Association for Computing Machinery: New York, NY, USA, 2021; pp. 594–595. [CrossRef]
15. Kazdaridis, G.; Sidiropoulos, N.; Zografopoulos, I.; Symeonidis, P.; Korakis, T. Nano-Things: Pushing Sleep Current Consumption to the Limits in IoT Platforms. In Proceedings of the 10th International Conference on the Internet of Things, Malmo, Sweden, 6–9 October 2020; Association for Computing Machinery: New York, NY, USA, 2020. [CrossRef]
16. Capra, M.; Peloso, R.; Masera, G.; Ruo Roch, M.; Martina, M. Edge Computing: A Survey On the Hardware Requirements in the Internet of Things World. *Future Int.* **2019**, *11*, 100. [CrossRef]
17. Mansouri, Y.; Babar, M. A review of edge computing: Features and resource virtualization. *J. Parallel Distrib. Comput.* **2021**, *150*, 155–183. [CrossRef]
18. Ghodsi, Z.; Garg, S.; Karri, R. Optimal checkpointing for secure intermittently-powered IoT devices. In Proceedings of the 2017 IEEE/ACM International Conference on Computer-Aided Design (ICCAD), Irvine, CA, USA, 13–16 November 2017; pp. 376–383. [CrossRef]
19. Mirhoseini, A.; Rouhani, B.D.; Songhori, E.; Koushanfar, F. Chime: Checkpointing Long Computations on Intermittently Energized IoT Devices. *IEEE Trans.-Multi-Scale Comput. Syst.* **2016**, *2*, 277–290. [CrossRef]
20. Gelenbe, E.; Siavvas, M. Minimizing Energy and Computation in Long-Running Software. *Appl. Sci.* **2021**, *11*, 1169. [CrossRef]
21. Jayakumar, H.; Raha, A.; Raghunathan, V. Energy-Aware Memory Mapping for Hybrid FRAM-SRAM MCUs in IoT Edge Devices. In Proceedings of the 2016 29th International Conference on VLSI Design and 2016 15th International Conference on Embedded Systems (VLSID), Kolkata, India, 4–8 January 2016; pp. 264–269. [CrossRef]
22. Ciuffoletti, A. An Open-source Testbed for IoT Systems. In Proceedings of the 17th International Conference on Web Information Systems and Technologies, Webist, 26–28 October 2021; pp. 397–403.
23. ESP8266EX Datasheet. Available online: https://www.espressif.com/sites/default/files/documentation/0a-esp8266ex_datasheet_en.pdf (accessed on 2 January 2020).