Data On the Go: Seamless Data Routing for Intermittently-Powered Battery-Free Sensing

Gaosheng Liu, *Member, IEEE,* Lin Wang, *Senior Member, IEEE*

Abstract—The rising demand for sustainable IoT has promoted the adoption of battery-free devices intermittently powered by ambient energy for sensing. However, the intermittency poses significant challenges in sensing data collection. Despite recent efforts to enable one-to-one communication, routing data across multiple intermittently-powered battery-free devices, a crucial requirement for a sensing system, remains a formidable challenge.

This paper fills this gap by introducing Swift, which enables seamless data routing in intermittently-powered battery-free sensing systems. Swift overcomes the challenges posed by device intermittency and heterogeneous energy conditions through three major innovative designs. First, Swift incorporates a reliable node synchronization protocol backed by number theory, ensuring successful synchronization regardless of energy conditions. Second, Swift adopts a low-latency message forwarding protocol, allowing continuous message forwarding without repeated synchronization. Finally, Swift features a simple yet effective mechanism for routing path construction, enabling nodes to obtain the optimal path to the sink node with minimum hops. We implement Swift and perform large-scale experiments representing diverse real-world scenarios. The results demonstrate that Swift achieves an order of magnitude reduction in end-to-end message delivery time compared with the state-of-the-art approaches for intermittently-powered battery-free sensing systems.

Index Terms—Battery-free sensing, intermittently-powered devices, network routing, sustainable IoT

I. Introduction

BATTERY-free (BF) devices have been increasingly adopted for sensing due to their maintenance-free and environment-friendly traits [1]–[8]. These devices harvest energy from ambient sources like light, vibrations, and radio waves [9]–[11], and buffer it in capacitors to function. However, the ambient energy is usually limited and cannot sustain continuous device operation. Consequently, BF devices alternate between short periods of activity (e.g., a few milliseconds) and much longer periods of inactivity (e.g., 100s of milliseconds or seconds) for recharging, following a charging cycle determined by ambient energy conditions. To overcome this limitation, recent efforts have focused on achieving continuous computing on a single BF device [12]–[21] and enabling one-to-one communication [22]–[24]. Despite these advances, a significant challenge remains unsolved: the issue of *data routing*, which involves transferring data across a network

of intermittently-powered BF devices, a crucial function for sensing data collection.

Efficient data routing faces significant challenges imposed by the *intermittency* of BF devices. In stark contrast to conventional duty cycling, where device states (e.g., active and sleep) are intentionally controlled [25]–[27], intermittency introduces *unpredictable* impacts on device operation. This uncertainty makes even simple tasks like maintaining reachability, which are straightforward for continuously-powered devices, much more complex. Synchronizing the working periods of two BF devices to ensure they are powered on simultaneously becomes a daunting task without knowledge of each other's charging cycle. Since BF devices spend most of their charging cycles turned off, the likelihood of both devices being powered on simultaneously is slim. Moreover, the heterogeneity of energy availabilities in different locations leads to diverse charging cycles for BF devices, requiring repeated synchronization attempts to maintain continuous reachability. Unfortunately, such synchronization efforts incur significant time overhead, proving detrimental for data routing, which relies on seamless, continuous data flow through the network, hop-by-hop.

Despite the extensive literature on routing [28], current solutions suffer from inefficiencies due to their inability to handle device intermittency, resulting in significant message delivery delays caused by frequent node synchronization. Moreover, BF devices are energy-constrained and lack powerful geo-location components commonly found in continuously-powered devices, which are instrumental in efficient routing [28]. While energy-harvesting wireless sensor networks are closely related [29], [30], their continuous operation based on active power management and duty cycling [25], [26], [31], [32] sets them apart from sensing systems built with BF devices that operate intermittently on a millisecond timescale. These factors make existing routing approaches either inapplicable or inefficient, emphasizing the need for novel solutions.

In this paper, we propose Swift, an efficient routing scheme that enables seamless data routing for sensing systems built with intermittently-powered BF devices. Swift incorporates three major innovations to address the challenges related to node synchronization, message forwarding, and route construction. First, Swift tackles the challenge of synchronizing BF devices under diverse energy conditions with Swift-sync, a reliable synchronization protocol inspired by number theory principles. Specifically, we transform the node synchronization problem into an instance of the linear congruential generator (LCG) problem. Swift-sync adjusts the working period of the sending BF device following the LCG and guarantees reliability by ensuring the full-cycle property of the LCG.

With Swift-sync, two BF devices without shared knowledge can reliably discover each other within a fixed time frame.

Second, Swift incorporates a low-latency message forwarding mechanism called Swift-forward, specifically designed to facilitate efficient message passing for BF devices along a given path. The main challenge lies in the repeated synchronization between neighboring nodes at each hop, which leads to prolonged message delivery latency. We observe that once two BF devices are synchronized with Swift-sync, they can maintain their synchronized state without the need for further expensive synchronizations. Building upon this finding, Swift-forward optimizes message forwarding by caching the charging cycle of the next hop at each node upon successful synchronization and tracking the offset between the node's working time and that of its next hop. With Swift-forward, each node can synchronize with its next hop and return to its original state, all within a single charging cycle time. The synchronization overhead is significantly reduced, thus greatly improving the efficiency of message forwarding.

Finally, Swift accomplishes route construction with a protocol called Swift-route, which relies on a flooding mechanism initiated by the sink node at its core. BF devices announce their distance (i.e., hop count) to the sink node to their neighbors and update their next hop if they receive an announcement with a smaller hop count. To achieve an organized hop count propagation, Swift-route employs a carefully crafted timing strategy that creates a layered structure based on each node's proximity to the sink node. With that, Swift-route ensures the optimal construction of routes for each node towards the sink in terms of hop count and is easy to implement.

Overall, we make the following contributions in this paper. After identifying the key challenges for seamless routing in BF sensing systems (§II), we present Swift where we

- propose a reliable node synchronization protocol Swift-sync, guaranteeing successes within a fixed amount of time, regardless of the energy conditions (§III).
- introduce a low-latency message forwarding mechanism tailored for intermittently-powered BF devices, significantly reducing the synchronization overhead (§IV).
- present a route construction protocol Swift-route for BF sensing systems, allowing each node to obtain a path to the sink with the least hop count (§V).
- implement Swift in OMNeT++ and perform large-scale experiments to validate its performance under various real-world scenarios (§VI). We have also tested part of Swift on a hardware testbed. Results show that Swift achieves an order of magnitude gains over the state-of-the-art approaches in reducing the message delivery time.

§VII discusses related works and §IX concludes the paper.

II. BACKGROUND AND MOTIVATION

In this section, we introduce the background on BF sensing, identify the challenges brought by BF devices in sensing systems, and motivate our work.

A. Intermittently-Powered BF Sensing

Advancements in energy-harvesting technologies have significantly spurred the adoption of BF devices for the develop-

ment of sensing systems, particularly in challenging environments [2], [5]–[7], [33]. These BF devices are equipped with energy harvesters capable of capturing ambient energy such as energy from the sun, vibrations, wind, and radio waves, which is then buffered in capacitors to power the device [11]. Since they operate passively without batteries, such sensing systems, once deployed, can last long without maintenance. Also, battery-freeness makes them more environment-friendly [2].

BF devices work *intermittently*—the device suffers power failures and alternates between on and off states constantly. This is because the harvested energy buffered in the tiny capacitor can only support the running of the device—a low-power microcontroller (e.g., TI MSP430-series)—for a very short period (e.g., a few milliseconds or less) [34]. On the other hand, charging the capacitor takes significantly longer (e.g., 100s of milliseconds or even seconds) due to the scarcity of ambient energy [24]. A BF device is powered on when the capacitor is charged to a certain level and suffers a power failure when the charging level of the capacitor drops below a threshold. We call the time when the device is on the *working period* and the *charging period* otherwise. A *charging cycle* spans one charging period and one working period.

BF sensing systems exhibit distinctive characteristics that differentiate them from traditional wireless sensor networks (WSNs) and energy-harvesting WSNs (EH-WSNs). The main differences between these systems are summarized in Table I. WSNs and EH-WSNs, typically powered by relatively rich energy supplies (e.g., rechargeable batteries and/or large capacitors), adopt active power management and duty cycling to control the device state (e.g., active, sleep) intentionally [25], [26], [31], [32]. On the other hand, BF sensing systems rely on devices powered by the scarce energy buffered in small capacitors (at the μF level) passively, resulting in an *intermittent* operational style at the millisecond scale induced by frequent power failures. This intermittency poses challenges for programs running on BF devices, as progress can be hindered if intermediate execution states are not preserved correctly. To address this issue, researchers have proposed two main categories of techniques: checkpointing program state [19], [21], [35] and transforming programs into idempotent tasks [13], [14], [36], [37]. Although these efforts focus on individual BF devices, they have established a solid foundation for the adoption of intermittently-powered BF devices in sensing applications.

B. Synchronizing Intermittently-Powered BF Devices

To establish communication between two intermittently-powered BF devices, they need to be in the active state simultaneously, necessitating the synchronization of their working periods. The challenge lies in the intermittent nature of their activity and the limited shared knowledge between the two devices, further complicating the synchronization process.

In scenarios where the energy condition is homogeneous across all devices, i.e., all BF devices operate on the same charging cycle, the problem becomes determining the offset between the working periods of two devices. To this end, Find utilizes a randomized approach to identify the working period

TABLE I: BF Sensing Systems vs. Traditional WSNs

	WSNs	Energy-Harvesting WSNs	Battery-Free Sensing
Energy storage	Batteries	Rechargeable batteries and/or large capacitors (mF/F)	Small capacitors (μF)
Power management	Active	Active	Passive
Operational style	Duty-cycling, predicable	Duty-cycling, predictable	Intermittent, unpredictable
Working time scale	Years	From minutes to hours	Milliseconds

offset between two devices [23]. In each charging cycle, the device employs a randomized approach to introduce a delay to its working period. This delay is generated from a well-tuned geometric distribution, allowing synchronization in a reasonable number of charging cycles on average. However, due to the random nature of this process, certain cases might require a significant or even infinite number of charging cycles for successful synchronization. In practice, BF devices are distributed geographically over a wide area. The heterogeneous energy conditions in different locations make it improbable for all BF devices to adhere to the same charging cycle. Consequently, ensuring constant reachability after synchronization becomes uncertain. To address this challenge, Bonito adopts a learning-based approach to predict the charging cycles based on statistical models [24]. In a nutshell, these approaches do not ensure successful synchronization, hence providing no reliability guarantee.

C. Data Routing for Intermittently-Powered BF Sensing

Typically, sensing systems collect data from various nodes and send it to a central sink node for processing and analysis. However, due to limited communication coverage, not all nodes can directly communicate with the sink. Consequently, messages generated by distant nodes need to be relayed through intermediate nodes to reach the sink. This challenge is the essence of the routing problem [28]. The main objective in routing is to find an optimal path that efficiently delivers messages to the sink in a timely manner.

The current routing protocols are not well-suited for intermittently-powered BF sensing due to the following reasons: (1) Existing protocols designed for WSNs or EH-WSNs rely on active duty cycling to regulate device state intentionally [28], [29]. However, this approach is unsuitable for intermittently-powered BF devices because they have limited power supply and unpredictable power outages. (2) Most routing protocols do not consider the time overhead involved in synchronizing two intermittently-powered BF devices, which results in high message delivery latency. (3) Protocols developed for continuously-powered devices often involve complex logic and utilize expensive geo-location peripherals, making them unaffordable for intermittently-powered BF devices.

Efficient data routing for intermittently-powered BF sensing involves addressing the following challenges:

1) *Minimizing node synchronization overhead:* To achieve efficient routing, messages need to be forwarded hop-by-hop on the path with minimal node synchronization overhead. Constantly synchronizing with the next hop for every transmission would be prohibitively costly for intermittently-powered BF devices.

2) *Optimal path selection:* Each node must be able to determine an efficient path towards the sink without knowing the location of other nodes. The goal is to find a path with the least number of hops required for the node to reach the sink. This way, data can be efficiently relayed to the central sink node.

In light of the above challenges, we aim to design a routing scheme specific to intermittently-powered BF sensing, featuring mechanisms for node synchronization, message forwarding, and route construction.

III. RELIABLE NODE SYNCHRONIZATION

In this section, we clarify the system model and introduce our synchronization algorithm based on the number theory.

A. System Model

We consider an intermittently-powered BF system comprising a fixed number of BF devices distributed across a large geographical area. These devices (nodes) possess identical hardware, including components like microcontrollers, capacitors, and energy-harvesting units, thus having roughly the same working period duration. We divide time into time slots where the working period for each node spans one time slot, while the charging period extends over multiple time slots since it is substantially longer than the working period as mentioned. The duration of the charging period, denoted as t, may differ among nodes based on their respective energy conditions. It might also fluctuate over time but usually at a much larger time scale (e.g., minutes). In practice, t usually falls within the range of $[5, 500]$ ms with capacitors of 10s of μF typically used on BF devices and the difference in charging time between any two nodes can be bounded by a constant in a given environment, denoted as α. In a typical stable environment without abnormal disruptions, α can be set to a small number [23], [24], and in our setting, we use $\alpha = 3$. This means the maximum charging time of a BF device is at most three times that of any other devices in the sensing system. Note that our proposed designs also work under more hostile environments with larger values for α, albeit with possibly reduced efficiency.

In this work, we assume a time slot lasts for one millisecond, as also seen in existing works [23], [24]. We argue that this period is long enough for successful communication (e.g., sending a message out and receiving an acknowledgment back) between a pair of battery-free devices if their working periods are perfectly aligned. For example, BLE 5.0 is capable of transmitting up to 175 bytes within one millisecond [38]. The slot length may deviate from 1 ms in real-world scenarios. As long as a round-trip communication between two BF devices
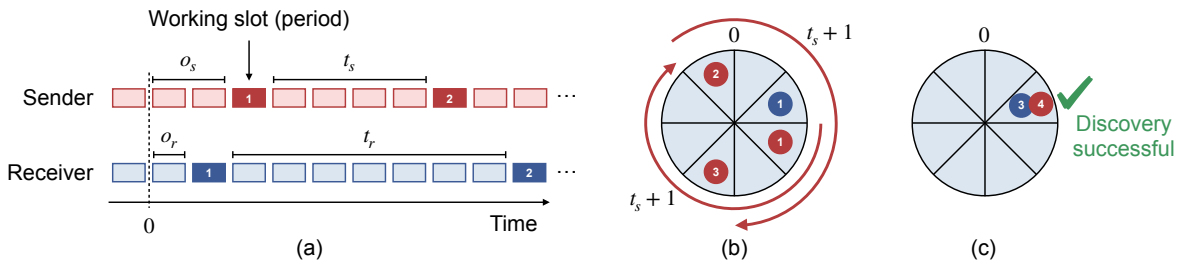
Fig. 1: Illustration of the node synchronization procedure: (a) charging cycles of the sender and the receiver over time (boxes represent time slots), (b) the offsets of the sender's working periods in relation to the receiver's charging cycle hypothetically, and (c) successful discovery when the working period offsets of the sender and receiver match.

can be completed within one slot, our mode will apply. However, achieving perfect alignment of time slots among battery-free devices is a challenging task in itself. Existing works have attempted to use an external light source that is perceivable at all battery-free devices for time slot alignment [23], which may not always be available in different sensing scenarios. We assume such a time slot alignment mechanism is used whenever it is feasible. Nevertheless, our model and solutions can be generalized to cases without external mechanisms for time slot alignment. Assume the communication between two BF devices can be successful when the overlap of the working periods of these devices exceeds 50% of one time slot (i.e., 500 μs), which is quite practical with BLE 5.0. By trying one more charging cycle, it is always ensured that two BF devices can communicate if we delay the working period of one of the BF devices by one time slot in the second charging cycle. Once one communication round is performed, the two BF devices can infer their time slot offset by monitoring the difference between their working time and communication time. For simplicity, we will use aligned time slots to describe our protocol design.

B. Theoretical Analysis

The Find approach shows statistically good node synchronization performance for cases with homogeneous energy conditions [23]. Extending it to heterogeneous cases is possible, but it has two main limitations: First, fine-tuning the randomization parameters for optimal performance becomes challenging, as parameters set for one neighbor may negatively affect others when a node synchronizes with multiple neighbors. Second, Find does not offer a guaranteed synchronization outcome due to its random nature. To address these limitations, we propose a deterministic approach that ensures a guaranteed success rate for node synchronization. Before presenting our design, we provide a theoretical analysis of the node synchronization problem.

For a pair of nodes trying to synchronize, we call the node sending out messages the *sender* and the one receiving messages the *receiver*. Figure 1 depicts the charging cycles of the sender and receiver, along with the synchronization procedure. We select a random time reference (time 0 shown in the figure) and denote the distance of the working period of a node from this reference as the initial offset. This initial offset is denoted by o_s and o_r for the sender and receiver,

TABLE II: Notation

Symbol	Meaning
o_s / o_r	Initial charging time offset for the sender/receiver
t_s / t_r	Charging time of the sender/receiver
c_inc	Current increment for co-prime
α	Maximum difference factor for the charging time
δ	Maximum increment for co-prime lookup
b_s / b_r	Bias of the sender working period with respect to the original working period of the sender/receiver
d_s	Delay to apply to the sender to remain synchronized
t_dis	Discovery time for a BF device to find a neighbor
t_self	Charging time of a BF device
t_max	Maximum charging time of a BF device
t_wait	Remaining listening (waiting) slot for the neighbors

respectively. Figure 1(a) shows that following the initial offset, the sender follows a charging cycle of length $t_s + 1$ while the receiver follows a charging cycle of $t_r + 1$. To facilitate understanding, we list all the notation throughout the paper in Table II.

One convenient way to represent the synchronization procedure is to choose the charging cycle of the receiver as a reference *hypothetically* and analyze the offsets of the working periods of the sender on the receiver's charging cycle, as shown in Figure 1(b). For the receiver, the working period offsets can be calculated as $(o_r + n \times (t_r + 1))\%(t_r + 1)$, with $n \in \mathbb{N}$, which is always equal to o_r. That is, the location of the working periods of the receiver in Figure 1 is fixed. For the sender, the offsets of its working periods with respect to the charging cycle of the receiver can be calculated as $(o_s + n \times (t_s + 1))\%(t_r + 1)$ which also provides the positions of the working periods of the sender in Figure 1. Intuitively, the sender cycles through the charging cycle of the receiver with a step of size $(t_s + 1)\%(t_r + 1)$.

The goal is to find a time slot where both the sender and the receiver are in the working state. For the example shown in Figure 1, the sender and receiver can communicate with each other in the fourth working period of the sender and the third working period of the receiver, as shown in Figure 1(c).

An important question to consider is whether synchronization is guaranteed to succeed within a fixed amount of time. The answer, however, is affirmative only in specific cases. In particular, we demonstrate that:

Theorem 1. *The node synchronization is guaranteed to be successful if $t_s + 1$ and $t_r + 1$ are coprime.*

Proof. We prove this by mapping the node synchronization problem to the linear congruential generator (LCG) problem. We refer readers to Chapter 3 of [39] for a detailed explanation of LCG. An LCG is an algorithm that produces a sequence of pseudo-randomized numbers defined by the recurrence relation:

$$X_{n+1} = (aX_n + c) \mod m, \tag{1}$$

where X is the sequence of the pseudo-random numbers, m is the modulus, and $a \in (0, m)$ and $c \in [0, m)$ the multiplier and the increment, respectively. $X_0 \in [0, m)$ is the seed for the sequence generation. For an LCG to obtain its maximum length m, a.k.a. full cycle LCG, it must satisfy the following conditions according to the Hull-Dobell Theorem [40]:

C1: c and m are coprime;
C2: $a - 1$ is divisible by all prime factors of m; and
C3: if 4 divides m, then 4 divides $a - 1$.

The node synchronization problem can be reduced to the LCG problem if we set $a = 1$, $c = t_s + 1$, and $m = t_r + 1$. X represents the offset of the working period of the sender in relation to the charging cycle (i.e., t_r+1) of the receiver. In this case, obtaining the maximum length for this LCG means that X will traverse all possible numbers in the range $[0, t_r + 1)$. This ensures that starting from seed $X_0 = o_s$, for some $n \in \mathbb{N}$ we will have $X_n = o_r$. In the context of the synchronization problem, this means that there will always be a case where the working period offsets of the sender and the receiver match with each other. This requires satisfying the above conditions for the Hull-Dobell Theorem where **C2** and **C3** are satisfied trivially and according to **C1** we require $t_s + 1$ and $t_r + 1$ to be coprime, hence completing the proof. $\qquad\square$

This result indicates that node synchronization is naturally guaranteed only when t_s+1 and t_r+1 are coprime. However, if this condition is not satisfied, the synchronization guarantee is lost, and more importantly, the nodes are not able to determine whether the condition holds or not in advance. Therefore, to ensure successful synchronization in all cases, a mechanism must be devised. In the following sections, we illustrate how we can achieve this objective by building upon Theorem 1 and introduce our proposed synchronization protocol.

C. Reliable Node Synchronization with Swift-sync

We propose Swift-sync, a reliable synchronization protocol with guaranteed success. Our idea is inspired by Theorem 1, where in cases of the coprime condition not being met, we come up with a simple trial-and-error strategy to create such a condition artificially. Note that our algorithm does not rely on any assumptions regarding the coprime condition, which will unlikely hold in real-world scenarios.

The idea of Swift-sync is as follows: We start by assuming that $c = t_s + 1$ and $m = t_r + 1$ are coprime and try to validate this assumption within a worst-case bound. If the synchronization is not successful within this bound, we increase c by one slot and repeat the above procedure. The worst-case bound is calculated as the maximum number of slots needed for the working period offsets of the sender and receiver to match, assuming c and m are coprime. Following

---

Algorithm 1 Node synchronization mechanism Swift-sync

1: **Input:** node's charging time: t_self, maximum charging cycle multiplier: α, maximum coprime gap: δ
2: **Output:** discovery time: t_dis, neighbor's ID: id_nbr, neighbor's charging time: t_nbr)
3: $c\_inc \leftarrow 0$, $cycle\_count \leftarrow 0$, $flag \leftarrow$ false
4: $id\_nbr \leftarrow -1$, $t\_nbr \leftarrow 0$, $t\_dis \leftarrow 0$
5: **while** $flag ==$ false **do**
6:     // Maximum tries reached, step forward by one slot
7:     **if** $cycle\_count > \alpha(t\_self + 1)$ **then**
8:         **if** $c\_inc > \delta$ **then**
9:             No neighbors available, return error
10:         **else**
11:             $c\_inc \leftarrow c\_inc + 1$
12:             $cycle\_count \leftarrow 0$
13:         **end if**
14:     **end if**
15:     $cycle\_count \leftarrow cycle\_count + 1$
16:     Delay the working period by c_inc time slots
17:     // Coprime condition met, working period offsets matched
18:     **if** neighbor found **then**
19:         $flag \leftarrow$ true
20:         $t\_dis \leftarrow (c\_inc + t\_self + 1)$
21:         Record neighbor's ID id_nbr and charging time t_nbr
22:     **end if**
23: **end while**

---

the LCG problem, for every charging cycle of the sender, the working period offset of the sender corresponds to a random number generated by the LCG. In the worst case, m random numbers need to be generated before we match with the working period offset of the receiver. Since the sender has no information about the charging cycle of the receiver in reality, we assume t_r is bounded by constant factor times of t_s where the constant is typically quite small, e.g., three. Combined with the charging time range of $[5, 500]$ slots, the upper bound for $t_s + 1$ is thus 1503 slots. Through our empirical observation, we notice that the maximum coprime gap for two numbers under 1503 is 10. By increasing c with a step of one slot upon synchronization failure, it is ensured that there will be a case where c and m are coprime within at most ten attempts, meeting the conditions for Theorem 1 and thus making the synchronization successful. Note that the assumptions of the charging time ranges are realistically derived from real-world conditions [23], [24].

The workings of Swift-sync are listed in Algorithm 1. We keep searching until a neighbor is synchronized (lines 5–23). In every round, we check if we have reached the maximum attempts for the current increment for c (line 7), i.e., c_inc which is assumed to be bounded by α times the charging cycle of the node itself, $\alpha(t\_self + 1)$. If so, we further check if we have reached the maximum necessary increment for c (line 8), which is bounded by the coprime gap δ. If that is true, we can conclude that there are no neighbors available and the synchronization algorithm terminates with an error (line 9). Within the maximum coprime gap, the increment grows by one (line 11) and we use the new increment to continue the search. In every charging cycle of the node, we increase the number of attempts (line 15) by one and delay the working period of the node by c_inc. If a neighbor node is found, we stop the synchronization process and return the discovery
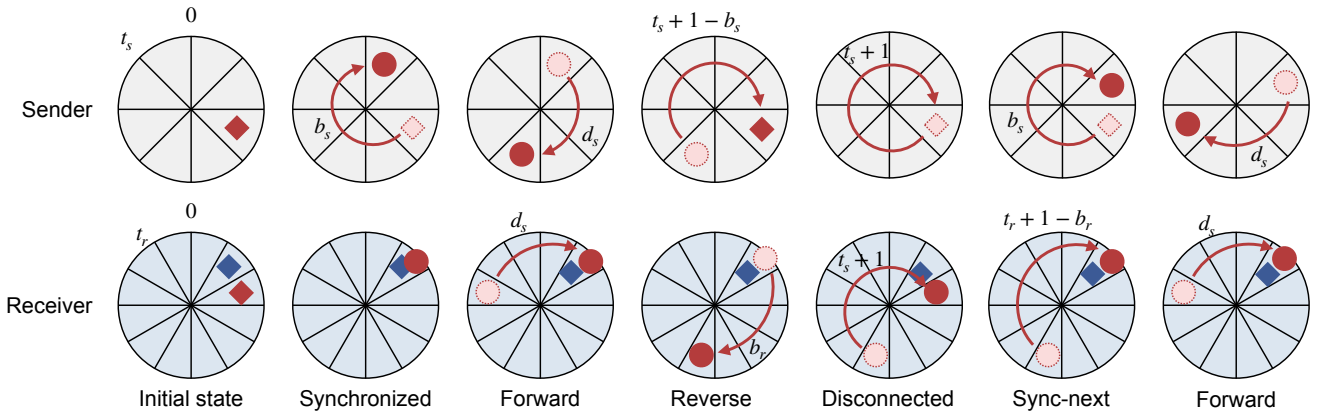
Fig. 2: Illustration of the state changes with the Swift-forward mechanism, where we show how the charging cycles of the sender should be adapted to synchronize with its next hop (e.g., the receiver) directly and reverse back when the message forwarding is completed. The receiver is passive and the sender uses the receiver's charging cycle as its reference.

time, the ID of the found neighbor, and the charging time of the found neighbor to facilitate future communications with this neighbor (lines 18–22). If no neighbors have been found after we have incremented c by more than δ, we can conclude that there are no reachable neighbors, and an error is returned. This is because Swift-sync guarantees that if a neighbor is available, it will always be found within the maximum number of attempts—a big advantage of a deterministic algorithm.

IV. MESSAGE FORWARDING

In a route towards the sink, each node must synchronize with its next hop before sending out messages with existing routing protocols, resulting in significant time overhead. However, we posit that node synchronization is only necessary for the initial communication with the next hop. Once synchronized, the node can retrieve and record the charging cycle information and the relative working period of the next hop. This allows for efficient communication during periods when the charging cycles remain stable, eliminating the need for repeated synchronizations for every hop on the route.

A. High-Level Idea

We explain the high-level idea through the example shown in Figure 2. We start with the initial state, where both the sender and receiver follow their own charging cycles and work periodically in time slots indicated with the colored diamond (red: sender, blue: receiver). Without loss of generality, we assume $t_s < t_r$ in this example. The zero point is just an arbitrary reference point. We assume the receiver is passive and thus, it does not adjust its working periods in the listening state. We denote by b_s the bias of the working period of the sender w.r.t. its initial working period position. Using the charging cycle of the receiver as a reference, the sender's working slot unlikely matches that of the receiver, and we denote by b_r the bias of the sender's working period w.r.t. the working period of the receiver, reflected on the receiver's charging cycle. Each node in the system can be in one of the following states, with state transitions depicted in Figure 2.

Synchronized During synchronization, the sender follows the Swift-sync algorithm (Algorithm 1) where b_s is updated as $b_s = (b_s + t_s + 1 + c\_inc)\%(t_s + 1)$. When the synchronization is completed, the sender's working period is aligned with that of the receiver and we set $b_r = 0$.

Forward Upon synchronization, the sender maintains the synchronized state with the receiver by delaying its working period by $d_s = (t_r - t_s)\%(t_r + 1)$ slot(s) in every charging cycle. As a result, b_s needs to be updated with $b_s = (b_s + d_s)\%(t_s + 1)$ and b_r remains zero. The sender remains synchronized with the receiver in every charging cycle and messages can be delivered between the two nodes.

Reverse Once the message forwarding is done, the sender reverses back its working period to its initial state so that it can be synchronized with its upstream nodes. Since b_s keeps a record of the bias of the current working period of the sender w.r.t. the initial value zero, we can simply delay the working period of the sender by $t_s + 1 - b_s$ slot(s) in the next charging cycle. As a result of that, we set $b_r = (t_s + 1 + t_s + 1 - b_s)\%(t_r + 1)$ where the first $(t_s + 1)$ component comes from the fact that one charging cycle is needed for applying the artificial delay and set $b_s = 0$.

Disconnected The reverse operation disconnects the two nodes. In this state, the sender may serve as a receiver for an upstream node. To be able to fast sync with its next hop without running Swift-sync again, the sender updates b_r with $b_r = (t_s + 1 + b_r)\%(t_r + 1)$ in each of its charging cycles.

Sync-next When the sender decides to forward messages to its next hop again, the sender simply delays its working period by $t_r + 1 - b_r$ slot(s), and the working periods of the two nodes become aligned. Here, we set $b_s = (t_r + 1 - b_r)\%(t_s + 1)$ and $b_r = 0$. After that, the sender enters the Forward state for delivering messages to the receiver, as already described above.

Algorithm 2 Message forwarding protocol Swift-forward

1: **Input:** charging time of the current node and its next hop: t_s, t_r, current node's working period bias w.r.t. its initial state b_s and its next hop b_r, id of the next hop: id_next
2: **procedure** SEND($queue$)
3:     $attempt \leftarrow 0$
4:     **while** $queue.size \neq 0$ **do**
5:         $d_s \leftarrow |t_r - t_s|\%(t_r + 1)$
6:         Delay the working period by d_s slot(s) in every cycle
7:         $msg \leftarrow queue.$pop(), $msg.id\_next \leftarrow id\_next$
8:         Send message msg and wait for ACK
9:         **if** ACK timeout (e.g., due to collision) **then**
10:             $attempt \leftarrow attempt + 1$, retry
11:             **if** $attempt > max\_attempt$ **then**
12:                 Break
13:             **end if**
14:         **end if**
15:         $b_s \leftarrow (b_s + d_s)\%(t_s + 1)$
16:         // Forwarding done, reverse to receiving
17:         **if** $queue.size == 0$ **then**
18:             Break
19:         **end if**
20:     **end while**
21:     $b_r \leftarrow (t_s + 1 + t_s + 1 - b_s)\%(t_r + 1)$
22:     $b_s \leftarrow 0$
23:     Reverse to the receiving state by delaying $(t_s + 1 - b_s)$ slot(s)
24: **end procedure**
25: **procedure** RECEIVE($queue$)
26:     $cycle\_wait \leftarrow 0$, $flag\_receive \leftarrow$ false
27:     **while** true **do**
28:         Listen and receive message msg
29:         **if** $msg.id\_next == self.id$ **then**
30:             $flag\_receive \leftarrow$ true
31:             $queue.$push(msg) and send ACK
32:             $b_r = (t_s + 1 + b_r)\%(t_r + 1)$
33:             **if** $msg.is\_last ==$ true **then**
34:                 $flag\_receive \leftarrow$ false
35:                 Break
36:             **end if**
37:         **end if**
38:         **if** $flag\_receive ==$ false **then**
39:             $cycle\_wait \leftarrow cycle\_wait + 1$
40:             **if** $cycle\_wait > max\_wait$ **then**
41:                 **if** $queue.size == 0$ **then**
42:                     Continue
43:                 **end if**
44:                 Break
45:             **end if**
46:         **end if**
47:     **end while**
48:     $b_s \leftarrow (t_r + 1 - b_r)\%(t_s + 1)$
49:     $b_r \leftarrow 0$
50:     Reverse to the sending state by delaying $t_r + 1 - b_r$ slot(s)
51: **end procedure**

B. Message Forwarding with the Swift-forward

Based on the above concepts, we introduce Swift-forward, a message-forwarding protocol that operates on each node and comprises two key procedures: *sending* and *receiving* during the steady state. In this protocol, nodes alternate their status between these two procedures, as detailed below.

1) Sending Procedure: The top of Algorithm 2 lists the sending procedure SEND($queue$) of the Swift-forward protocol. For a given node, we assume it is already in sync with its next hop and has entered the Forward state. If the queue (for caching messages to be sent out) is not empty, the node maintains the synchronized state with its next hop (lines 5–6). In every charging cycle, the node pops a message from the queue, sets the metadata of the message, and sends the message out (lines 7–8). If it times out before receiving the ACK (e.g., due to collisions), it will keep retrying until the maximum number of attempts reaches an upper threshold and the SEND procedure aborts directly (lines 9–14). Meanwhile, it updates b_s as we explained before (line 15). When the queue becomes empty, the sending procedure is terminated and the node reverses its state and starts to follow the receiving procedure (lines 17–19). The node also resets b_r and b_s as explained in the Reverse state (lines 21–23). In case the next hop cannot be reached after a few charging cycles, the sender concludes that the charging cycle of the sender or receiver, or both has changed and calls Swift-sync to re-synchronize with its next hop.

2) Receiving Procedure: The bottom of Algorithm 2 lists the receiving procedure RECEIVE($queue$) of the Swift-forward protocol. If the node is synchronized with a previous hop, the node listens and receives incoming messages (line 28). If the incoming message is valid, the node pushes the message to its queue and sends an acknowledgment back (lines 29–31). The node also updates its working period bias with respect to its own next hop b_r, where in every charging cycle, the node advances $(t_s + 1 + b_r)\%(t_r + 1)$ slot(s) on the charging cycle of its next hop (line 32). Upon the last message from the synchronized previous hop, the node leaves the connection (lines 33–36) and then reverses to the sending procedure directly. If the expected receiving is not successful, the node counts down its waiting cycles. When the number of waiting cycles reaches the maximum (empirically set to $\alpha(t_s + 1)$) (lines 40–45), the node performs the following depending on the queue status: If there are no messages in its queue, the node keeps waiting (line 42); otherwise, the node switches to the sending state (line 44). As a result, the node applies Sync-next to synchronize its working time with its next hop and resets b_s and b_r as explained in the Sync-next state (lines 48–50).

V. ROUTE CONSTRUCTION

We propose a route construction protocol Swift-route, which is simple to implement and ensures each node reaches the sink with the minimum hops. The workings of Swift-route are listed in Algorithm 3. Each node runs two procedures: MSGHANDLER and BROADCAST. The MSGHANDLER procedure is responsible for processing incoming messages. Upon receiving a message, we check if a shorter path has been found with the received message. If so, we keep the shorter path and update the hop count (lines 5–9) and wait for a certain amount of time (i.e., t_wait cycles of t_max slots each) announced by the message source. During the waiting period, if another message with an even shorter path is received, we update the hop count and restart the waiting timer with the waiting time contained in the new message (lines 10–13). Upon timeout, the node broadcasts its hop count with the BROADCAST procedure.

The BROADCAST procedure performs maximum attempts to ensure that the message of a node can reach all the node's

---

Algorithm 3 Route construction protocol Swift-route

1: **Input:** node's id: id_self, node's charging time and its upper-bound: t_self, t_max, maximum coprime gap: δ
2: **Output:** node's next hop: id_next
3: $cur\_hop \leftarrow \infty$, $t\_wait \leftarrow 0$, $id\_next \leftarrow$ null, $c\_inc \leftarrow 0$
4: **procedure** MSGHANDLER(msg)
5:      **if** $cur\_hop > msg.hop + 1$ **then**
6:          $id\_next \leftarrow msg.id$
7:          $cur\_hop \leftarrow msg.hop + 1$
8:          $t\_wait \leftarrow msg.t\_wait$
9:      **end if**
10:      **while** $t\_wait > 0$ **do**
11:          $t\_wait \leftarrow t\_wait - 1$
12:          Call MSGHANDLER() upon new messages
13:      **end while**
14:      BROADCAST(id_self, cur_hop)
15: **end procedure**
16: **procedure** BROADCAST(id_self, cur_hop)
17:      // Maximum attempts to ensure all neighbors are reached
18:      **while** $c\_inc < \delta$ **do**
19:          $cycle\_count \leftarrow 0$
20:          **while** $cycle\_count < t\_max$ **do**
21:              $cycle\_count \leftarrow cycle\_count + 1$
22:              $t\_wait \leftarrow t\_max(\delta - c\_inc) - cycle\_count$
23:              Delay working period by c_inc
24:              Broadcast $\langle id\_self, cur\_hop, t\_wait \rangle$
25:          **end while**
26:          $c\_inc \leftarrow c\_inc + 1$
27:      **end while**
28: **end procedure**

Fig. 3: Illustration for a contradiction where we show that hop D-F cannot exist on any constructed route.

potential neighbors (lines 18–27). We assume the maximum charging period is t_max long and hence, the maximum number of rounds is bounded by $\delta \cdot t\_max$ where δ is the maximum coprime gap. The waiting time necessary for the message receiver to wait before they broadcast is the time left for the current node to finish its broadcast (line 22). This is to avoid a downstream node starting to broadcast its hop count before it hears from all possible upstream nodes. In every charging cycle, the node delays its working period by c_inc and sends out the hop count message together with the expected waiting time, hoping to reach a potential neighbor. c_inc is incremented until it reaches δ to ensure a coprime to the neighbor's charging cycle is covered for sure. Therefore, the following theorem holds.

Theorem 2. *Swift-route ensures that the hop count of the path from each node to the sink is minimized.*

Proof. We define the height of a node as the least hop count to the sink. Since Swift-route always keeps the path with the least hop count upon receiving messages, the proof can be conducted by showing that a node can't choose its next hop with the same height. We consider the contradiction scenario shown in Figure 3. Let us assume hop D-F is part of the path of a node. This is only possible if node F has received the broadcast from node D and has started its own broadcasting before receiving the broadcast from C (hence it missed the broadcast message from C). This means that node F must have waited for D to finish its broadcasting and node D must have also waited for B to finish its broadcasting. Since in our route construction protocol, we set the maximum broadcast time of a
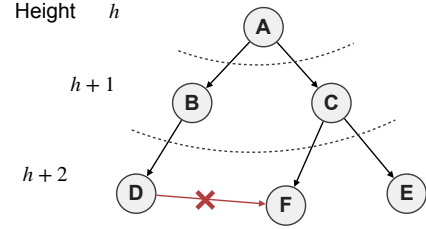
node to the bound $t\_bcast = (t\_max+1) \cdot \delta \cdot t\_max$ slots. This is based on the worst case where the node has the maximum charging time in the system. Hence, node F has to wait at least $2 \times t\_bcast$ slots after B starts to broadcast before it can start broadcasting. However, with $2 \times t\_bcast$, node C is guaranteed to have finished its broadcasting (since B and C are at the same height and thus require the same amount of time for finishing broadcasting), which contradicts that node F has not received the broadcast from C. This completes the proof. $\square$

VI. PERFORMANCE EVALUATION

We implement Swift in OMNeT++ and perform extensive experiments with a variety of real-world parameter setups and compare Swift with state-of-the-art approaches retrofitted with intermittency support. We choose OMNeT++ since it allows us to focus on the key aspects of the protocol and assess the performance on large scales. In the following, we first explain our evaluation setups. For Swift-sync we also implement a prototype and test it on a hardware testbed. Then, we compare Swift with baselines and study the impact of different environmental factors.

A. Experimental Setups

Following findings in existing works [23], [24], we assume the charging time t of all nodes falls into the range of $[5, 500]$ time slots where a time slot is set to 1ms. Additionally, for two nodes i and j, we assume their charging time difference is bounded by a small constant as mentioned before, i.e., $t_j/\gamma \le t_i \le \gamma t_j$ where $\gamma \in \{2, 3, 4\}$. We consider three ranges of node charging time, $[5, 15]$ for good energy conditions, $[40, 120]$ for medium energy conditions, and $[166, 498]$ for poor energy conditions. For evaluating the node synchronization protocol, we use the public dataset for real-world charging times of BF devices [23] and set γ to three.

We consider two cases for the sensing system configuration: *square* and *rectangle*. In the square case, 100 nodes are randomly distributed in a square area of 45 m $\times$ 45 m, while the same number of nodes are randomly distributed in a rectangle area of 25 m $\times$ 100 m in the rectangle case. In both cases, the sink node sits in the middle of the area's right edge. These two cases show how Swift performs under different environments with a varying average number of hops from each node to the sink. We also vary BF devices' communication range (10m, 15m, 20m) to analyze its impact on the routing performance.
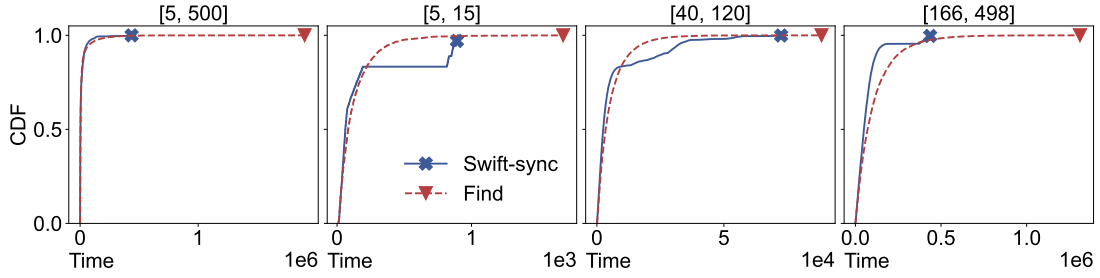
Fig. 4: Node synchronization time (in slots) with Swift-sync and Find [23] under varying energy condition ranges. The markers denote the 99th percentile.
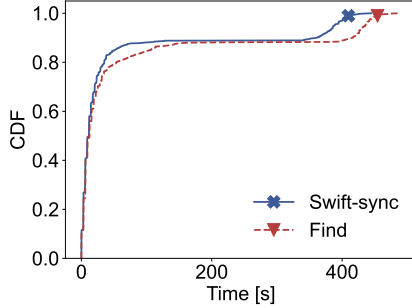

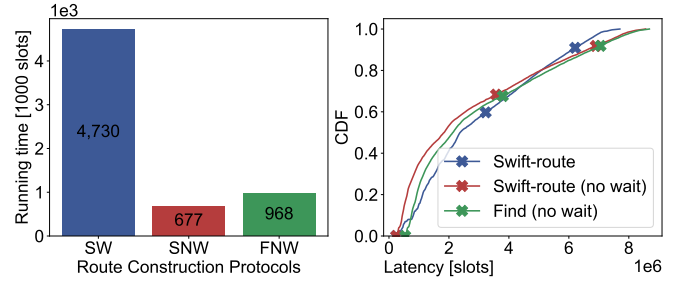Fig. 5: Node synchronization time on the hardware prototype.

Fig. 6: (left) route construction time (in slots) of Swift-route with waiting (SW), Swift-route without waiting (SNW), and Find without waiting (FNW), and (right) end-to-end message delivery time CDF of the three protocols.

Our baselines are chosen according to the evaluation goals, as detailed in the following subsections.

B. Performance of Swift-sync

We first evaluate the performance of Swift-sync by comparing it with Find [23], which is the state-of-the-art node synchronization mechanism between BF devices. We adapt Find to scenarios with heterogeneous energy conditions. We consider two nodes to be synchronized. With Find nodes delay for a random number of slots following a geometric distribution with fine-tuned parameters. We consider all three energy conditions (i.e., good, medium, and poor) where we filter out the charging time pairs (each for one of the nodes) provided in a public dataset [23] falling into these energy conditions. Due to the randomization of Find, for each pair of charging times, we repeat simulations 1000 times to avoid noise.

Figure 4 depicts the cumulative distribution function (CDF) of the node synchronization time (in slots) with Swift-sync and Find under the three energy conditions as well as the aggregated results. We observe that Swift-sync outperforms Find in three aspects: (1) Under all three energy conditions, Swift-sync achieves lower synchronization time than Find at around the 80th percentile. We can also observe the long tail of the latency CDF of Find, demonstrating the uncertainty caused by randomness. (2) Under good and median energy conditions (ranges $[5, 15]$ and $[40, 120]$), Swift-sync outperforms Find under around the 80th percentile, while Swift-sync is better than Find under around 98th percentile under poor energy conditions (range $[166, 498]$). Compared with Find, Swift-sync reduces the average synchronization time by more than 20%.

To further validate Swift-sync, we develop a prototype with two intermittently powered BF devices. The BF device consists of three parts: a control unit, a power unit, and a communication unit. Both the control and power units are built with TI-MSP430FR5994 boards, while the communication unit is built with a NORDIC-NRF52840 board for BLE communication. The power and control units are interconnected with UART and the control unit talks to the BLE unit through SPI. The power unit controls a MOSFET to supply power to the control unit, emulating different intermittency behaviors in the power supply. With this setup, we tested the performance of Swift-sync under the charging time range $[40, 120]$ slots where a slot lasts for 41 ms based on our hardware setup. The much higher slot length is due to the overheads incurred by the communication between multiple boards to emulate an intermittently-powered BF device. Nonetheless, our protocol still applies. Figure 5 shows that the result matches that from the simulation. We can see that Swift-sync outperforms Find in most cases. Overall, Swift-sync achieves efficient synchronization while ensuring reliability.

C. Performance of Swift-route

We now evaluate the performance of Swift-route. A major feature of Swift-route is its optimality regarding the hop count of each node to the sink. This is achieved by the waiting mechanism in the design of the protocol. We thus compare with a variant of Swift-route where this waiting mechanism is disabled, i.e., each node broadcasts its hop count immediately after they receive a message containing a smaller hop count.
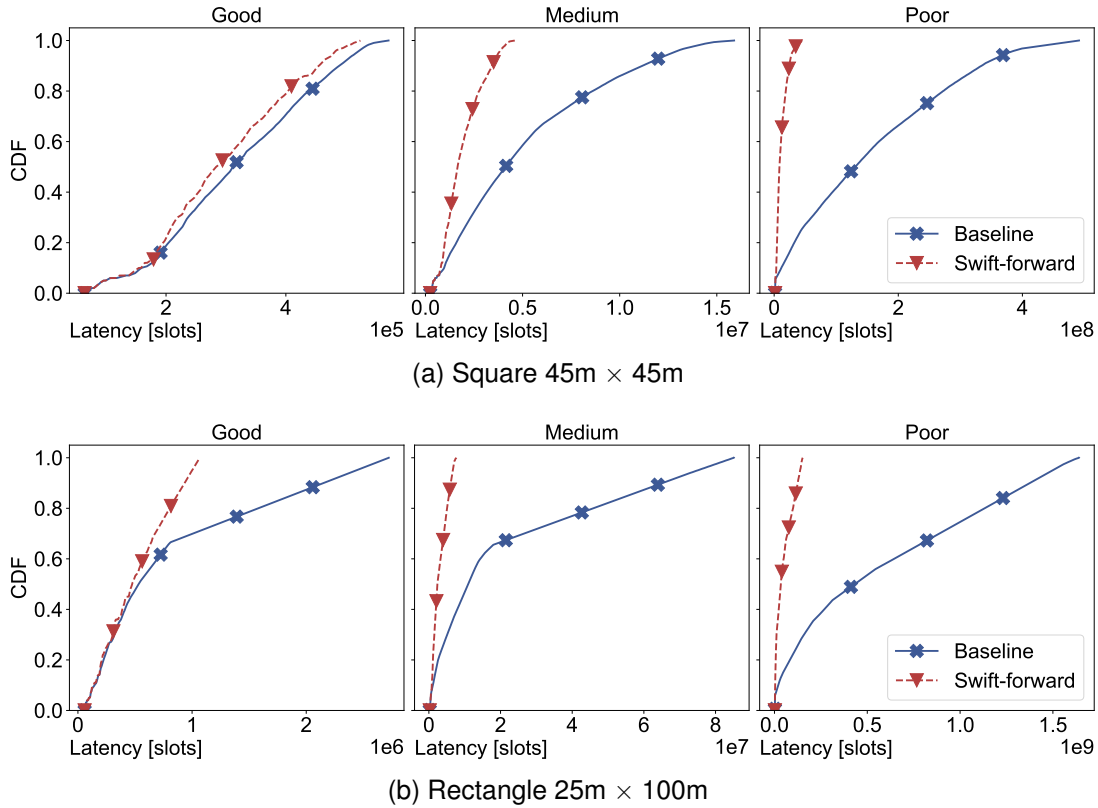
Fig. 7: Message forwarding time (in slots) comparison between Swift-forward and a baseline without fast synchronization under varying energy conditions and two field scenarios.
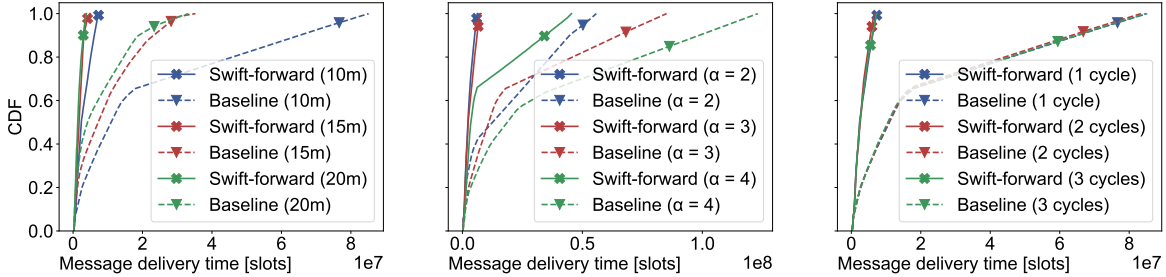

Fig. 8: Performance of Swift under varying parameter setups: (left figure) the communication range, (middle figure) the maximum charging time difference, and (right figure) the message intensity.

We call this variant Swift-route (no-wait). We also compare with a baseline where the node synchronization is done with Find, without the waiting mechanism. We call this baseline Find (no-wait). Traditional routing protocols for WSNs are not considered due to their inability to handle intermittency.

Figure 6(left) shows the time (in slots) required to complete the route construction protocol on the rectangle field under the medium energy condition. As expected, Swift-route takes the longest time, due to the waiting mechanism in the protocol to ensure optimality. Considering that the protocol is only needed to run to establish the routes at the beginning, this time overhead is acceptable. While taking far less time to complete, the baselines produce suboptimal paths, leading to long delays in message delivery. Figure 6(right) shows the message delivery time CDF when using the routes generated

by each of the three approaches. We can see that Swift-route can largely reduce the message delivery time for nodes with long routes compared with the baselines thanks to the shorter paths for these nodes. However, this is achieved at the expense of higher message delivery time for nodes with short routes between them. One major reason is that the shortest paths produced by Swift form the shortest path tree, with all other links not on any shortest paths pruned out, leading to the worst path diversity. This is likely to create more congestion on some links compared with the baselines where different links may be used to spread out the load.

D. Performance of Swift-forward

To evaluate Swift-forward, we compare it with a baseline where we disable fast synchronization in Swift-forward. In

other words, the baseline employs the node synchronization mechanism every time a node communicates with its next hop. We evaluate all three energy conditions (good, medium, poor) and with the two areas (square and rectangle).

Figure 7 shows the CDF results for the end-to-end message delivery time (in slots) under the above setups. As we can observe, Swift-forward is highly efficient in message forwarding, where the end-to-end message delivery time is reduced by up to an order of magnitude under all the considered conditions. Comparing the results for the different energy conditions (good, medium, and poor), we see a clear trend where the gap between Swift-forward and the baseline becomes larger when the energy condition improves, suggesting that Swift-forward can maintain its efficiency even in hostile environments.

E. Impact of Parameters

We also test Swift-forward under different setups to examine the impact of system parameters on the message forwarding performance. We focus on the end-to-end message delivery time (in slots) as the performance metric and choose the same baseline (without fast synchronization) as we used in the previous experiment.

Figure 8 shows the performance of Swift-forward under varying parameter setups. The left figure depicts the *impact of the communication range* where we consider each node can communicate directly with other nodes within a range of 10m, 15m, and 20m, respectively. We can see that Swift-forward performs better than the baseline in all cases. Moreover, when the communication range increases, the end-to-end message delivery time decreases as expected, which is attributed to the reduced hop count for each node to reach the sink. The middle figure depicts the *impact of the maximum charging time difference* (i.e., α), where we show three cases with $\alpha = 2$, $\alpha = 3$, and $\alpha = 4$, respectively. Similarly, Swift-forward outperforms the baseline in all cases. With the maximum charging time difference increase, the end-to-end message delivery time generally increases. This can be explained by the fact that the node synchronization takes more time with a larger charging time difference between nodes. Finally, the right figure shows the *impact of the message intensity* in the network. We consider three message intensity levels where each node sends a message every one, two, and three cycles, respectively. Again, Swift-forward beats the baseline in all cases as expected. We do not observe much performance difference in these cases, which indicates that Swift-forward is equally efficient under all the considered message intensities.

The selection of these parameters highly depends on the specific application requirements and deployment conditions. In general, it is always beneficial to increase the communication range of BF devices, if the BF devices are equipped with capacitors large enough to sustain the communication in the working period of the device. The charging time range is usually dictated by the environment. However, it is generally beneficial to avoid deploying BF devices at locations with extremely bad energy availability to achieve an α as small as possible.

VII. RELATED WORKS

Communication for BF devices. Communication between BF devices has gained significant attention, and the first challenge is addressing the node synchronization problem. Without shared knowledge, node synchronization is attempted by delaying the working period of one or both devices following specific distributions [23], [41], [42], hoping they align after several rounds. These approaches require fine-tuning based on system parameters and energy conditions and guarantee no success due to their randomized nature. Bonito proposes to learn the charging time with well-known distributions to enable continuous communication [24]. De Winkel et al. also present a Bluetooth stack for battery-free devices without changing the protocol specification [22]. While focusing on a single hop, these works provide a solid foundation for the routing problem we study in this paper. In short, existing routing protocols fall short because of (1) unrealistic assumptions not holding in intermittent sensing systems, making them inapplicable, and (2) not accounting for the high synchronization overhead in intermittent sensing systems, making them inefficient.

Low-power and lossy wireless networks. Significant research has been conducted on low-power and lossy wireless networks (LLWNs), including time-slotted channel hopping (TSCH) [43] and routing protocols for low power and lossy networks (RPL) [44]. Within this domain, concepts such as frame-slots and sub-slots have been introduced. Notably, sub-slots are subsets of frame-slots, typically spanning a 10-millisecond scale [45]–[48]. However, this does not imply that sensor nodes operate within this time frame. Instead, coordinators utilize these time slots to schedule communications, effectively reducing collision risks among sensor nodes. In contrast, intermittently-powered networks present unique challenges. Here, the active working time of BF devices is measured at a millisecond scale. Predicting their charging time is complex, and for this study, we consider a simplified scenario. Moreover, unlike traditional networks where a coordinator manages synchronization, in intermittently-powered sensing systems, BF devices independently handle their synchronization processes.

Routing in sensing systems. Routing is an essential function of any sensing system and has been heavily studied in WSNs. Assuming continuous power supply by the onboard battery, traditional WSNs often leverage potent devices that are not available for intermittently-powered BF sensing systems [49], [50]. Meanwhile, energy-harvesting WSNs employ energy harvesters to recharge the onboard battery or super-capacitors and adopt active duty cycling to control the device state intentionally [25], [26], [29], [31], [32], [51], [52], [52]–[54]. Both traditional WSNs and energy-harvesting WSNs assume more powerful energy storage than intermittently-powered sensing systems.

Linear congruential generator. The Linear Congruential Generator (LCG) is a widely recognized method for generating random numbers and has been a staple in the realm of computational algorithms. LCG plays a crucial role in security communications and is frequently employed in wireless sensor networks for random number generation. To our knowledge,

LCG has been utilized for encrypting data in several studies, as indicated in references [55]–[57]. Additionally, EDGF [58] has leveraged LCG for generating datasets in simulation environments. Notably, Swift represents the pioneering work in using a customized LCG for synchronizing BF devices, representing a significant advancement in this field.

VIII. DISCUSSION

A. Comparative Analysis with EH-WSNs

Traditional wireless sensor networks (WSNs) typically consist of sensor nodes that are powered by chemical batteries [49]. The routing problem has been studied extensively, but the goal is mainly to pursue QoS and the whole network life through balancing the energy consumption of nodes. In light of the maintenance challenge, researchers have explored the idea of energy-harvesting WSNs (EH-WSNs), where the sensor node is powered by a rechargeable battery or a supercapacitor [59]. With this difference, the goal of routing optimization turns to the QoS and the uptime of the network. The uptime of the network depends on the energy availability which is dictated by the energy harvesting rate and the battery/capacitor size. Most of the existing works leverage prediction models to assess the energy availability and perform energy management, based on which the routing schemes are developed [59]–[61]. Intermittently-powered battery-free sensing systems are fundamentally different from EH-WSNs since the buffered energy in the capacitor will be depleted in just one wake-up of the BF device, lasting only one or a few milliseconds. Also, the charging time is much shorter due to the limited capacity of the tiny capacitor. Therefore, there is no energy management needed across device wake-ups unlike in EH-WSNs. Our routing scheme is specially tailored for intermittently powered BF sensing systems, with the goal of minimizing the number of wake-ups needed for device synchronization and message passing.

B. Scalability

Scaling intermittently powered sensing systems remains a challenge in general due to the high level of dynamism and unpredictability the environment may impose on the system. While taking a significant step, Swift is still limited by the assumption it builds on: the charging time of a BF device stays stable at a small time scale (e.g., a few minutes). This assumption typically holds for deployment in indoor environments (e.g., a greenhouse or warehouse) where the environment is highly controlled and more stable than outdoor wild environments. For applications in a greenhouse or warehouse, we think that the size we are targeting (i.e., 100 BF devices used in the evaluation) is reasonable. For indoor environments on even larger scales, Swift can still work, albeit the performance (w.r.t. node synchronization time and end-to-end message delivery time) may degrade inevitably due to the increase of the hop count in the topology. One mitigation strategy is to increase the communication range of BF devices to reduce the hop count. However, this will require equipping these BF devices with larger capacitors, thus negatively affecting the form factor of the device as well as the deployment cost. For more hostile conditions where the charging time of BF devices varies dramatically in an unpredictable fashion, Swift will unlikely work as expected, and future research work is needed.

C. Energy Efficiency

Energy efficiency is an important topic for embedded systems like the sensing system we consider in this paper. Nevertheless, we would like to point out that there are fundamental differences between intermittently powered battery-free devices and traditional battery-powered sensor devices. With traditional battery-powered sensor devices, the energy saved from an efficient protocol design can be used for other tasks of the system. However, intermittently powered battery-free devices wake up when enough energy has been collected and stored in the onboard capacitor with an artificial delay imposed by the protocol. In every device wake-up, the device only performs a fixed communication round. As a result, the energy efficiency of the protocol can mainly be characterized by the number of device wake-ups needed for synchronization and message passing. This means that the performance metrics we focus on in the evaluation, e.g., the synchronization time, the route construction latency, and the end-to-end message delivery time, already capture the energy efficiency aspect of the protocol. In short, the faster the protocol is, the more energy efficient the protocol is, which means that fewer charging cycles will be spent on the protocol and more cycles on other tasks.

IX. CONCLUSIONS

This paper studies routing with intermittency and proposes Swift—a routing scheme tailored for intermittently-powered BF sensing systems. Swift features three new designs: a node synchronization protocol Swift-sync enabling the communication between BF devices with success guarantee, a message forwarding protocol Swift-forward that avoids repeated synchronization between nodes on the route to achieve low latency, and a route construction mechanism Swift-route that produces a route from each node to the sink node with the least hop count. Our evaluation based on an implementation in OMNeT++ and real-world setups shows that Swift achieves efficient routing and outperforms the state-of-the-art to a large extent. In future work, we aim to explore scenarios involving higher temporal variations in charging time.

REFERENCES

[1] J. D. Hester and J. Sorber, "The future of sensing is batteryless, intermittent, and awesome," in *Proceedings of the 15th ACM Conference on Embedded Network Sensor Systems, SenSys 2017, Delft, Netherlands, November 06-08, 2017*, M. R. Eskicioglu, Ed. ACM, 2017, pp. 21:1–21:6. [Online]. Available: https://doi.org/10.1145/3131672.3131699

[2] B. Lucia, V. Balaji, A. Colin, K. Maeng, and E. Ruppel, "Intermittent computing: Challenges and opportunities," in *2nd Summit on Advances in Programming Languages, SNAPL 2017, May 7-10, 2017, Asilomar, CA, USA*, ser. LIPIcs, B. S. Lerner, R. Bodík, and S. Krishnamurthi, Eds., vol. 71. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2017, pp. 8:1–8:14. [Online]. Available: https://doi.org/10.4230/LIPIcs.SNAPL.2017.8

[3] H. Desai, M. Nardello, D. Brunelli, and B. Lucia, "Camaroptera: A long-range image sensor with local inference for remote sensing applications," *ACM Trans. Embed. Comput. Syst.*, vol. 21, no. 3, pp. 32:1–32:25, 2022. [Online]. Available: https://doi.org/10.1145/3510850

[4] T. Li and X. Zhou, "Battery-free eye tracker on glasses," in *Proceedings of the 10th on Wireless of the Students, by the Students, and for the Students Workshop, S3@MobiCom 2018, New Delhi, India, November 2, 2018*, S. Pradhan and S. K. Saha, Eds. ACM, 2018, pp. 27–29. [Online]. Available: https://doi.org/10.1145/3264877.3264885

[5] G. Liu and L. Wang, "Self-sustainable cyber-physical systems with collaborative intermittent computing," in *e-Energy '21: The Twelfth ACM International Conference on Future Energy Systems, Virtual Event, Torino, Italy, 28 June - 2 July, 2021*, H. de Meer and M. Meo, Eds. ACM, 2021, pp. 316–321. [Online]. Available: https://doi.org/10.1145/3447555.3465324

[6] A. Y. Majid, P. Schilder, and K. Langendoen, "Continuous sensing on intermittent power," in *19th ACM/IEEE International Conference on Information Processing in Sensor Networks, IPSN 2020, Sydney, Australia, April 21-24, 2020*. IEEE, 2020, pp. 181–192. [Online]. Available: https://doi.org/10.1109/IPSN48710.2020.00-36

[7] M. Afanasov, N. A. Bhatti, D. Campagna, G. Caslini, F. M. Centonze, K. Dolui, A. Maioli, E. Barone, M. H. Alizai, J. H. Siddiqui, and L. Mottola, "Battery-less zero-maintenance embedded sensing at the mithræum of circus maximus," in *SenSys '20: The 18th ACM Conference on Embedded Networked Sensor Systems, Virtual Event, Japan, November 16-19, 2020*, J. Nakazawa and P. Huang, Eds. ACM, 2020, pp. 368–381. [Online]. Available: https://doi.org/10.1145/3384419.3430722

[8] K. Geissdoerfer, I. Splitt, and M. Zimmerling, "Demo abstract: Building battery-free devices with riotee," in *The 22nd International Conference on Information Processing in Sensor Networks, IPSN 2023, San Antonio, TX, USA, May 9-12, 2023*. ACM, 2023, pp. 354–355. [Online]. Available: https://doi.org/10.1145/3583120.3589808

[9] V. Talla, B. Kellogg, S. Gollakota, and J. R. Smith, "Battery-free cellphone," *Proc. ACM Interact. Mob. Wearable Ubiquitous Technol.*, vol. 1, no. 2, pp. 25:1–25:20, 2017. [Online]. Available: https://doi.org/10.1145/3090090

[10] J. de Winkel, V. Kortbeek, J. D. Hester, and P. Pawelczak, "Battery-free game boy: Sustainable interactive devices," *GetMobile Mob. Comput. Commun.*, vol. 25, no. 2, pp. 22–26, 2021. [Online]. Available: https://doi.org/10.1145/3486880.3486888

[11] A. Colin, E. Ruppel, and B. Lucia, "A reconfigurable energy storage architecture for energy-harvesting devices," in *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2018, Williamsburg, VA, USA, March 24-28, 2018*, X. Shen, J. Tuck, R. Bianchini, and V. Sarkar, Eds. ACM, 2018, pp. 767–781. [Online]. Available: https://doi.org/10.1145/3173162.3173210

[12] J. van der Woude and M. Hicks, "Intermittent computation without hardware support or programmer intervention," in *12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016, Savannah, GA, USA, November 2-4, 2016*, K. Keeton and T. Roscoe, Eds. USENIX Association, 2016, pp. 17–32. [Online]. Available: https://www.usenix.org/conference/osdi16/technical-sessions/presentation/vanderwoude

[13] A. Colin and B. Lucia, "Chain: tasks and channels for reliable intermittent programs," in *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2016, part of SPLASH 2016, Amsterdam, The Netherlands, October 30 - November 4, 2016*, E. Visser and Y. Smaragdakis, Eds. ACM, 2016, pp. 514–530. [Online]. Available: https://doi.org/10.1145/2983990.2983995

[14] K. S. Yildirim, A. Y. Majid, D. Patoukas, K. Schaper, P. Pawelczak, and J. D. Hester, "Ink: Reactive kernel for tiny batteryless sensors," in *Proceedings of the 16th ACM Conference on Embedded Networked Sensor Systems, SenSys 2018, Shenzhen, China, November 4-7, 2018*, G. S. Ramachandran and B. Krishnamachari, Eds. ACM, 2018, pp. 41–53. [Online]. Available: https://doi.org/10.1145/3274783.3274837

[15] H. A. Asad, E. H. Wouters, N. A. Bhatti, L. Mottola, and T. Voigt, "On securing persistent state in intermittent computing," in *Proceedings of the 8th International Workshop on Energy Harvesting & Energy-Neutral Sensing Systems, ENSsys@SenSys 2020, Virtual Event, Japan, November 16, 2020*. ACM, 2020, pp. 8–14. [Online]. Available: https://doi.org/10.1145/3417308.3430267

[16] M. Surbatovich, B. Lucia, and L. Jia, "Towards a formal foundation of intermittent computing," *Proc. ACM Program. Lang.*, vol. 4,

no. OOPSLA, pp. 163:1–163:31, 2020. [Online]. Available: https://doi.org/10.1145/3428231

[17] K. Maeng and B. Lucia, "Adaptive low-overhead scheduling for periodic and reactive intermittent execution," in *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2020, London, UK, June 15-20, 2020*, A. F. Donaldson and E. Torlak, Eds. ACM, 2020, pp. 1005–1021. [Online]. Available: https://doi.org/10.1145/3385412.3385998

[18] S. Ahmed, N. A. Bhatti, M. H. Alizai, J. H. Siddiqui, and L. Mottola, "Fast and energy-efficient state checkpointing for intermittent computing," *ACM Trans. Embed. Comput. Syst.*, vol. 19, no. 6, pp. 45:1–45:27, 2020. [Online]. Available: https://doi.org/10.1145/3391903

[19] V. Kortbeek, K. S. Yildirim, A. Bakar, J. Sorber, J. D. Hester, and P. Pawelczak, "Time-sensitive intermittent computing meets legacy software," in *ASPLOS '20: Architectural Support for Programming Languages and Operating Systems, Lausanne, Switzerland, March 16-20, 2020*, J. R. Larus, L. Ceze, and K. Strauss, Eds. ACM, 2020, pp. 85–99. [Online]. Available: https://doi.org/10.1145/3373376.3378476

[20] V. Kortbeek, A. Bakar, S. Cruz, K. S. Yildirim, P. Pawelczak, and J. D. Hester, "Bfree: Enabling battery-free sensor prototyping with python," *Proc. ACM Interact. Mob. Wearable Ubiquitous Technol.*, vol. 4, no. 4, pp. 135:1–135:39, 2020. [Online]. Available: https://doi.org/10.1145/3432191

[21] V. Kortbeek, S. Ghosh, J. D. Hester, S. Campanoni, and P. Pawelczak, "Wario: efficient code generation for intermittent computing," in *PLDI '22: 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation, San Diego, CA, USA, June 13 - 17, 2022*, R. Jhala and I. Dillig, Eds. ACM, 2022, pp. 777–791. [Online]. Available: https://doi.org/10.1145/3519939.3523454

[22] J. de Winkel, H. Tang, and P. Pawelczak, "Intermittently-powered bluetooth that works," in *MobiSys '22: The 20th Annual International Conference on Mobile Systems, Applications and Services, Portland, Oregon, 27 June 2022 - 1 July 2022*, N. Bulusu, E. Aryafar, A. Balasubramanian, and J. Song, Eds. ACM, 2022, pp. 287–301. [Online]. Available: https://doi.org/10.1145/3498361.3538934

[23] K. Geissdoerfer and M. Zimmerling, "Bootstrapping battery-free wireless networks: Efficient neighbor discovery and synchronization in the face of intermittency," in *18th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2021, April 12-14, 2021*, J. Mickens and R. Teixeira, Eds. USENIX Association, 2021, pp. 439–455. [Online]. Available: https://www.usenix.org/conference/nsdi21/presentation/geissdoerfer

[24] ——, "Learning to communicate effectively between battery-free devices," in *19th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2022, Renton, WA, USA, April 4-6, 2022*, A. Phanishayee and V. Sekar, Eds. USENIX Association, 2022, pp. 419–435. [Online]. Available: https://www.usenix.org/conference/nsdi22/presentation/geissdoerfer

[25] P. H. Kindt and S. Chakraborty, "On optimal neighbor discovery," in *Proceedings of the ACM Special Interest Group on Data Communication, SIGCOMM 2019, Beijing, China, August 19-23, 2019*, J. Wu and W. Hall, Eds. ACM, 2019, pp. 441–457. [Online]. Available: https://doi.org/10.1145/3341302.3342067

[26] R. Margolies, G. Grebla, T. Chen, D. Rubenstein, and G. Zussman, "Panda: Neighbor discovery on a power harvesting budget," vol. 34, no. 12, 2016, pp. 3606–3619. [Online]. Available: https://doi.org/10.1109/JSAC.2016.2611984

[27] Z. Dong, Y. Gu, J. Chen, S. Tang, T. He, and C. Liu, "Enabling predictable wireless data collection in severe energy harvesting environments," in *2016 IEEE Real-Time Systems Symposium, RTSS 2016, Porto, Portugal, November 29 - December 2, 2016*. IEEE Computer Society, 2016, pp. 157–166. [Online]. Available: https://doi.org/10.1109/RTSS.2016.024

[28] R. A. Uthra and S. V. K. Raja, "Qos routing in wireless sensor networks - a survey," *ACM Comput. Surv.*, vol. 45, no. 1, pp. 9:1–9:12, 2012. [Online]. Available: https://doi.org/10.1145/2379776.2379785

[29] K. S. Adu-Manu, N. H. Adam, C. Tapparello, H. Ayatollahi, and W. B. Heinzelman, "Energy-harvesting wireless sensor networks (eh-wsns): A review," *ACM Trans. Sens. Networks*, vol. 14, no. 2, pp. 10:1–10:50, 2018. [Online]. Available: https://doi.org/10.1145/3183338

[30] R. V. Prasad, V. S. Rao, C. Sarkar, and I. G. Niemegeers, "Renew: A practical module for reliable routing in networks of energy-harvesting wireless sensors," *IEEE Trans. Green Commun. Netw.*, vol. 5, no. 3, pp. 1558–1569, 2021. [Online]. Available: https://doi.org/10.1109/TGCN.2021.3094771

[31] J. Marasevic, C. Stein, and G. Zussman, "Max-min fair rate allocation and routing in energy harvesting networks: Algorithmic analysis," vol. 78, no. 2, 2017, pp. 521–557. [Online]. Available: https://doi.org/10.1007/s00453-016-0171-6

[32] Q. Chen, H. Gao, Z. Cai, L. Cheng, and J. Li, "Energy-collision aware data aggregation scheduling for energy harvesting sensor networks," in *2018 IEEE Conference on Computer Communications, INFOCOM 2018, Honolulu, HI, USA, April 16-19, 2018*. IEEE, 2018, pp. 117–125. [Online]. Available: https://doi.org/10.1109/INFOCOM.2018.8486366

[33] G. Gobieski, B. Lucia, and N. Beckmann, "Intelligence beyond the edge: Inference on intermittent embedded systems," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2019, Providence, RI, USA, April 13-17, 2019*, I. Bahar, M. Herlihy, E. Witchel, and A. R. Lebeck, Eds. ACM, 2019, pp. 199–213. [Online]. Available: https://doi.org/10.1145/3297858.3304011

[34] S. Ahmed, M. Nawaz, A. Bakar, N. A. Bhatti, M. H. Alizai, J. H. Siddiqui, and L. Mottola, "Demystifying energy consumption dynamics in transiently powered computers," *ACM Trans. Embed. Comput. Syst.*, vol. 19, no. 6, pp. 47:1–47:25, 2020. [Online]. Available: https://doi.org/10.1145/3391893

[35] K. Maeng and B. Lucia, "Supporting peripherals in intermittent systems with just-in-time checkpoints," in *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2019, Phoenix, AZ, USA, June 22-26, 2019*, K. S. McKinley and K. Fisher, Eds. ACM, 2019, pp. 1101–1116. [Online]. Available: https://doi.org/10.1145/3314221.3314613

[36] J. de Winkel, C. D. Donne, K. S. Yildirim, P. Pawelczak, and J. D. Hester, "Reliable timekeeping for intermittent computing," in *ASPLOS '20: Architectural Support for Programming Languages and Operating Systems, Lausanne, Switzerland, March 16-20, 2020*, J. R. Larus, L. Ceze, and K. Strauss, Eds. ACM, 2020, pp. 53–67. [Online]. Available: https://doi.org/10.1145/3373376.3378464

[37] J. D. Hester, K. M. Storer, and J. Sorber, "Timely execution on intermittently powered batteryless sensors," in *Proceedings of the 15th ACM Conference on Embedded Network Sensor Systems, SenSys 2017, Delft, Netherlands, November 06-08, 2017*, M. R. Eskicioglu, Ed. ACM, 2017, pp. 17:1–17:13. [Online]. Available: https://doi.org/10.1145/3131672.3131673

[38] M. Woolley, "Bluetooth® core specification version 5.0 feature enhancements," 2021. [Online]. Available: https://www.bluetooth.com/bluetooth-resources/bluetooth-5-go-faster-go-further

[39] D. E. Knuth, *The Art of Computer Programming, Volume 2: Seminumerical Algorithms*, 3rd ed. Addison-Wesley, 1997.

[40] J. L. Allard, A. R. Dobell, and T. E. Hull, "Mixed congruential random number generators for decimal machines," *J. ACM*, vol. 10, no. 2, pp. 131–141, 1963. [Online]. Available: https://doi.org/10.1145/321160.321163

[41] M. M. I. Rajib and A. Nasipuri, "Predictive retransmissions for intermittently connected sensor networks with transmission diversity," *ACM Trans. Embed. Comput. Syst.*, vol. 17, no. 1, pp. 12:1–12:25, 2018. [Online]. Available: https://doi.org/10.1145/3092947

[42] V. Deep, M. L. Wymore, A. A. Aurandt, V. Narayanan, S. Fu, H. Duwe, and D. Qiao, "Experimental study of lifecycle management protocols for batteryless intermittent communication," in *IEEE 18th International Conference on Mobile Ad Hoc and Smart Systems, MASS 2021, Denver, CO, USA, October 4-7, 2021*. IEEE, 2021, pp. 355–363. [Online]. Available: https://doi.org/10.1109/MASS52906.2021.00052

[43] S. Kharb and A. Singhrova, "A survey on network formation and scheduling algorithms for time slotted channel hopping in industrial networks," *J. Netw. Comput. Appl.*, vol. 126, pp. 59–87, 2019. [Online]. Available: https://doi.org/10.1016/j.jnca.2018.11.004

[44] J. V. V. Sobral, J. J. P. C. Rodrigues, R. A. L. Rabêlo, J. Al-Muhtadi, and V. Korotaev, "Routing protocols for low power and lossy networks in internet of things applications," *Sensors*, vol. 19, no. 9, p. 2144, 2019. [Online]. Available: https://doi.org/10.3390/s19092144

[45] N. Choudhury, M. M. Nasralla, P. Gupta, and I. U. Rehman, "Centralized graph based TSCH scheduling for iot network applications," in *2021 IEEE Intl Conf on Parallel & Distributed Processing with Applications, Big Data & Cloud Computing, Sustainable Computing & Communications, Social Computing & Networking (ISPA/BDCloud/SocialCom/SustainCom), New York City, NY, USA, September 30 - Oct. 3, 2021*. IEEE, 2021, pp. 1639–1644. [Online]. Available: https://doi.org/10.1109/ISPA-BDCloud-SocialCom-SustainCom52081.2021.00219

[46] L. Tang, Y. Sun, O. Gurewitz, and D. B. Johnson, "EM-MAC: a dynamic multichannel energy-efficient MAC protocol for wireless sensor networks," in *Proceedings of the 12th ACM International Symposium on Mobile Ad Hoc Networking and Computing, MobiHoc 2011, Paris, France, May 16-20, 2011*. ACM, 2011, p. 23. [Online]. Available: https://doi.org/10.1145/2107502.2107533

[47] D. D. Guglielmo, A. Seghetti, G. Anastasi, and M. Conti, "A performance analysis of the network formation process in IEEE 802.15.4e TSCH wireless sensor/actuator networks," in *IEEE Symposium on Computers and Communications, ISCC 2014, Funchal, Madeira, Portugal, June 23-26, 2014*. IEEE Computer Society, 2014, pp. 1–6. [Online]. Available: https://doi.org/10.1109/ISCC.2014.6912607

[48] S. Jeong, H. Kim, J. Paek, and S. Bahk, "OST: on-demand TSCH scheduling with traffic-awareness," in *39th IEEE Conference on Computer Communications, INFOCOM 2020, Toronto, ON, Canada, July 6-9, 2020*. IEEE, 2020, pp. 69–78. [Online]. Available: https://doi.org/10.1109/INFOCOM41043.2020.9155496

[49] C. T. Nakas, D. Kandris, and G. Visvardis, "Energy efficient routing in wireless sensor networks: A comprehensive survey," *Algorithms*, vol. 13, no. 3, p. 72, 2020. [Online]. Available: https://doi.org/10.3390/a13030072

[50] M. Zimmerling, L. Mottola, and S. Santini, "Synchronous transmissions in low-power wireless: A survey of communication protocols and network services," *ACM Comput. Surv.*, vol. 53, no. 6, pp. 121:1–121:39, 2021. [Online]. Available: https://doi.org/10.1145/3410159

[51] Y. Gu, T. Zhu, and T. He, "ESC: energy synchronized communication in sustainable sensor networks," in *Proceedings of the 17th annual IEEE International Conference on Network Protocols, 2009. ICNP 2009, Princeton, NJ, USA, 13-16 October 2009*, H. Schulzrinne, K. K. Ramakrishnan, T. G. Griffin, and S. V. Krishnamurthy, Eds. IEEE Computer Society, 2009, pp. 52–62. [Online]. Available: https://doi.org/10.1109/ICNP.2009.5339699

[52] L. Huang and M. J. Neely, "Utility optimal scheduling in energy-harvesting networks," *IEEE/ACM Trans. Netw.*, vol. 21, no. 4, pp. 1117–1130, 2013. [Online]. Available: https://doi.org/10.1109/TNET.2012.2230336

[53] T. Zhu, J. Li, H. Gao, and Y. Li, "Data aggregation scheduling in battery-free wireless sensor networks," *IEEE Trans. Mob. Comput.*, vol. 21, no. 6, pp. 1972–1984, 2022. [Online]. Available: https://doi.org/10.1109/TMC.2020.3035671

[54] S. Yang, Y. Tahir, P. Chen, A. Marshall, and J. A. McCann, "Distributed optimization in energy harvesting sensor networks with dynamic in-network data processing," in *35th Annual IEEE International Conference on Computer Communications, INFOCOM 2016, San Francisco, CA, USA, April 10-14, 2016*. IEEE, 2016, pp. 1–9. [Online]. Available: https://doi.org/10.1109/INFOCOM.2016.7524475

[55] A. D. Dhruva, B. Prasad, S. Kamepalli, S. Kunisetti *et al.*, "An efficient mechanism using iot and wireless communication for smart farming," *Materials Today: Proceedings*, vol. 80, pp. 3691–3696, 2023.

[56] B. Sun, Y. Xiao, C. Li, H. Chen, and T. A. Yang, "Security co-existence of wireless sensor networks and RFID for pervasive computing," *Comput. Commun.*, vol. 31, no. 18, pp. 4294–4303, 2008. [Online]. Available: https://doi.org/10.1016/j.comcom.2008.05.035

[57] K. Haseeb, I. U. Din, A. Almogren, and N. Islam, "An energy efficient and secure iot-based WSN framework: An application to smart agriculture," *Sensors*, vol. 20, no. 7, p. 2081, 2020. [Online]. Available: https://doi.org/10.3390/s20072081

[58] D. K. Sah, K. Cengiz, P. K. Donta, V. N. Inukollu, and T. Amgoth, "EDGF: empirical dataset generation framework for wireless sensor networks," *Comput. Commun.*, vol. 180, pp. 48–56, 2021. [Online]. Available: https://doi.org/10.1016/j.comcom.2021.08.017

[59] K. S. Adu-Manu, N. H. Adam, C. Tapparello, H. Ayatollahi, and W. B. Heinzelman, "Energy-harvesting wireless sensor networks (eh-wsns): A review," *ACM Trans. Sens. Networks*, vol. 14, no. 2, pp. 10:1–10:50, 2018. [Online]. Available: https://doi.org/10.1145/3183338

[60] D. Wu, J. He, H. Wang, C. Wang, and R. Wang, "A hierarchical packet forwarding mechanism for energy harvesting wireless sensor networks," *IEEE Commun. Mag.*, vol. 53, no. 8, pp. 92–98, 2015. [Online]. Available: https://doi.org/10.1109/MCOM.2015.7180514

[61] K. Geissdoerfer, R. Jurdak, B. Kusy, and M. Zimmerling, "Getting more out of energy-harvesting systems: energy management under time-varying utility with preact," in *Proceedings of the 18th International Conference on Information Processing in Sensor Networks, IPSN 2019, Montreal, QC, Canada, April 16-18, 2019*, M. R. Eskicioglu, L. Mottola, and B. Priyantha, Eds. ACM, 2019, pp. 109–120. [Online]. Available: https://doi.org/10.1145/3302506.3310393
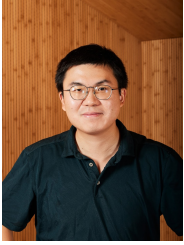
Gaosheng Liu is currently a PhD candidate at Vrije Universiteit Amsterdam. He received his M.S. degree in the College of intelligence and computing, Tianjin University, Tianjin, China, in 2019. He received his bachelor degree from Tianjin university of technology and education, Tianjin, China, in 2015. His research interests include intermittent communication, cyber-physical systems, and AIoT.

Lin Wang is currently a Full Professor and Head of the Computer Networks group in the Department of Computer Science at Paderborn University, Germany. He received his Ph.D. in Computer Science from the Institute of Computing Technology, Chinese Academy of Sciences in 2015. Before joining Paderborn University, he was a tenured Assistant Professor at Vrije Universiteit Amsterdam and held positions at TU Darmstadt, SnT Luxembourg, and IMDEA Networks Institute. His research is focused on networked systems, with the goal of achieving efficiency, scalability, and sustainability. He has served as a referee for several funding agencies (e.g., DFG, ISF, and HK-RGC), on the program committees of major conferences (e.g., SoCC, Middleware, ICCD, INFOCOM, MobiHoc, ICDCS, IWQoS, and SEC) and as a reviewer for top journals (e.g., ToN, JSAC, TMC, and TPDS). He has received several important awards, including a Google Research Scholar Award, an Outstanding Paper Award of IEEE RTSS 2022, Best Paper Awards of IEEE IPCCC 2023, IEEE ISCC 2024, and IEEE HotPNS 2016, and an Athene Young Investigator Award of TU Darmstadt. He is currently a Senior Member of IEEE.