Lure: a Simulator for Networks of Batteryless Intermittent Nodes

Mathew L. Wymore[a], Rohit Sahu[a], Thomas Ruminski[a], Vishal Deep[a], Morgan Ambourn[a], Gregory Ling[a], Vishak Narayanan[a], William Asiedu[a], Daji Qiao[a], Henry Duwe[a,*]

[a]*Iowa State University Department of Electrical and Computer Engineering, Coover Hall, 50010, Ames, USA*

Abstract

The emerging paradigm of batteryless intermittent sensor networks (BISNs) presents new challenges for researchers of low-power wireless systems and protocols. The nature of these challenges exacerbates the difficulty of evaluating networks of physical sensor nodes, making simulation an even more important component in evaluating performance metrics, such as communication throughput and delay, for BISN designs. To our knowledge, existing simulators and analytical models do not meet the unique needs of BISN research; therefore, we have created a new open-source BISN simulator named *Lure*. Lure is designed from the ground-up for simulation of batteryless intermittent systems and networks. Written in Python, Lure is powerful, flexible, highly configurable, and supports rapid prototyping of new protocols, systems, and applications, with a low learning curve. In this paper, we present Lure and validate it with experimental data to show that Lure can accurately reflect the reality of BISNs. We then demonstrate the process of applying Lure to research questions in select case studies.

Keywords: batteryless intermittent systems, intermittent communication, sensor networks, simulators

1. Introduction

Ubiquitous sensing and computing require pervasive deployment of billions of wireless IoT devices [1]. Powering such devices from batteries is both expensive and environmenally unsustainable [2, 3]. Batteryless devices [4] that harvest energy from the ambient environment, including from radio frequency (RF), solar, vibration and thermal sources, offer a promising solution to this problem [5–7]. Such devices leverage capacitors to buffer the harvested energy. As capacitance increases, charging time, power delivery overheads (e.g., equivalent resistances and charge leakage), and physical size increase. Therefore, pervasive batteryless sensor nodes are limited to small capacitances [6]. Often due to uncontrollable and weak harvested power compared to the operating power, these devices regularly become unpowered—i.e., they operate intermittently [8].

Intermittent operation of even a single batteryless node leads to challenges such as loss of volatile state, loss of time information, and forward progress guarantees. Solutions such as [9–16], [17–19], and [20, 21] have been proposed. There have been a range of single-node systems developed and deployed, but each is limited to applications with direct communication to a continuously-powered system [14, 22, 23]. However, to achieve the tantalizingly pervasive deployments batteryless systems purport, they must be networked.

Networking these nodes to form batteryless intermittent sensor networks (BISNs) is a critical component for the success of batteryless intermittent systems, but doing so presents a series of challenges that have just begun to be explored. One obstacle is that evaluation of protocols for intermittent systems is even more challenging than traditional WSNs and IoT systems. First, implementation of intermittent systems is more complex, on both the hardware and software sides. Second, experimental evaluation of intermittent systems takes more time to get the same number of data points, because nodes spend most of the time charging and not, for example, sending and receiving packets. Third, no current analytic models for intermittent systems

*Corresponding author

Email addresses: `mlwymore@iastate.edu` (Mathew L. Wymore), `rsahu@iastate.edu` (Rohit Sahu), `ruminski@iastate.edu` (Thomas Ruminski), `vdeep@iastate.edu` (Vishal Deep), `mambourn@iastate.edu` (Morgan Ambourn), `gling@iastate.edu` (Gregory Ling), `vishakn@iastate.edu` (Vishak Narayanan), `wasiedu@iastate.edu` (William Asiedu), `daji@iastate.edu` (Daji Qiao), `duwe@iastate.edu` (Henry Duwe)

are able to capture the tight coupling of an individual node's dynamic energy harvesting and the several layers of communication protocols needed to network BISNs.

Therefore, to enable fast prototyping and evaluation of protocols for intermittent networks, we need a simulator. An ideal BISN simulator would produce realistic results for intermittent networks that could serve as guidelines for physical system design, allow more convenient study of protocols before physical implementation and experimentation, and serve as a complementary tool to a physical implementation. However, existing simulators do not natively simulate the specialized hardware and software components of intermittent systems. Add-ons and modifications to these simulators have provided some support for BISNs, but this support is generally limited or protocol-specific. Over the course of our research, we have developed a novel SimPy-based simulator designed from the ground up for BISNs. We call this simulator *Lure* (because in fishing, a lure is a type of artificial bait that simulates the behavior of real bait, allowing the angler to get results with less overhead). In this paper, we publish Lure's source code[1].

In addition to releasing Lure as an open-source tool, the contributions of this paper include:

- A validation and evaluation of Lure using experimental results obtained from real prototype hardware.
- Three case studies demonstrating some of Lure's potential uses:
  1. Implementation and evaluation of a novel communication protocol that resides at a unique intermittency-aware netstack layer not present in existing simulators.
  2. Exploration of multihop networking performance at varying degrees of intermittency.
  3. Performance evaluation of a traditional time synchronization protocol in an intermittent context.

In Section 2, we present the background of batteryless intermittent systems including an intermittency aware netstack. In Section 3, we motivate the need for a BISN simulator by considering related work in network simulators and intermittent performance models, and demonstrating the insufficiency of the state-of-the-art intermittent communication model. In Section 4, we provide a brief description of Lure (our simulator for BISNs) followed by experimental validation of Lure's simulation capabilities. Finally, in Section 5, we present a series of case studies that depict Lure's potential as a tool for BISN research, before concluding the paper in Section 6. For additional reference, a detailed technical description of Lure is included as Appendix A.

[1]All simulator source code, case study scripts, and results are available at `https://zenodo.org/records/11062224` in the hopes that Lure can aid the intermittent systems research community in making intermittent networks a reality.

2. Batteryless Intermittent Systems Background

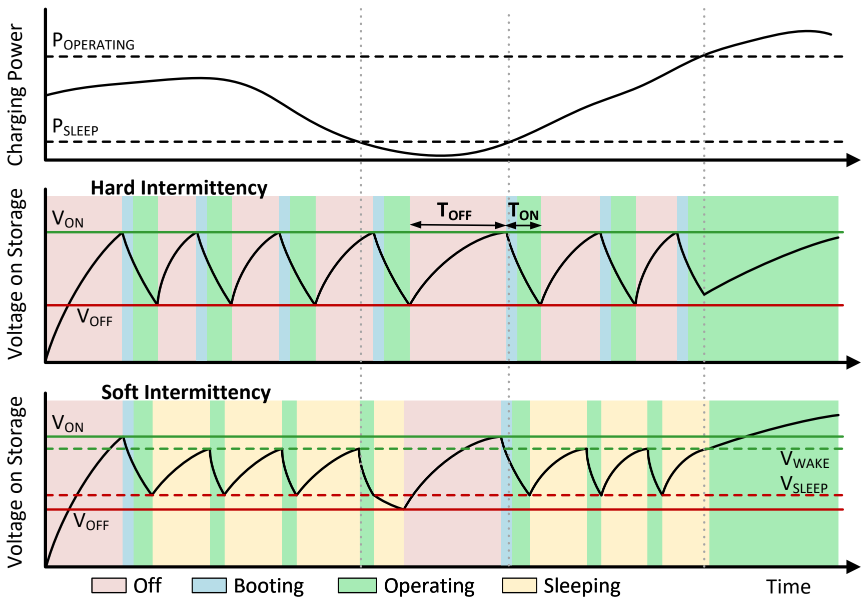


Figure 1: Illustration of intermittency. $P_{\text{OPERATING}}$ and P_{SLEEP} are the operating and sleep powers of the node, V_{ON} is the voltage at which the node turns on, V_{OFF} is the voltage at which the node turns off, T_{OFF} is the duration of an *off-time*, T_{ON} is the duration of an *on-time*, V_{SLEEP} is the voltage at which the node enters deep sleep (for soft intermittency), and V_{WAKE} (for soft intermittency) is the voltage at which the node returns to the operating state after sleeping.

2.1. Definitions

Batteryless intermittent systems are distinct from traditional WSN nodes in that they use a capacitor for energy storage instead of a battery, and this capacitor is relatively small in capacity (from hundreds of μF to a few tens of mF [20, 24, 25]) and physical size (from a few tens of mm in diameter and length [26]). This means all energy must come from the environment through harvesting. As illustrated by Figure 1, ambient energy sources are often weak compared to the node's operating power, causing nodes to operate *intermittently*, turning off to charge (*off-time*, T_{OFF}) and turning on only when sufficient energy is available to operate for some duration (*on-time*, T_{ON}, typically < 1 s).

To distinguish this off/on cycling of the entire node from the traditional concept of duty-cycling, we refer to it as *lifecycling*. We define the lifecycle ratio (LCR) of a hard intermittent node as the fraction of time which it spends powered on:

$$\text{LCR} = \frac{T_{ON}}{T_{OFF} + T_{ON}}. \tag{1}$$

The LCR of a node depends on how energy-rich the node's environment is. A more energy-rich environment yields a higher LCR, and vice versa, for a given operating power. Environmental conditions can vary, leading to interesting BISN research areas such as reliability in the presence of such variation. We also note that the on-time and off-time in Eq. (1) are coupled (e.g. a shorter on-time leads to a proportionally shorter off-time), so a node is not able to directly influence its LCR outside of changing its operating power.

2.2. Hard vs. Soft Intermittency

Intermittency can be managed through a variety of hardware and software architectures. This heterogeneity is one of the reasons a ground-up BISN simulator is useful. In general, intermittency can be categorized into two distinct flavors—*hard* and *soft*—illustrated in Fig. 1. In hard intermittency, when a node runs out of energy for operating, it *dies*; that is, it becomes completely unpowered at the microcontroller level. This presents some fundamental challenges, including:

- Loss of volatile program state, addressed through techniques such as checkpointing [9–12, 20] and task-based execution [13–16].
- Loss of active clock timing information, addressed using persistent clocks [17–19].
- Loss of precise control over when a node is on, because the duration of off-times depends on ambient energy harvesting [24, 25].

In contrast, soft intermittency attempts to avoid off-time (i.e. avoid lifecycling) by entering a deep sleep state before running out of energy. If the harvesting rate is higher than the power consumption of the node in the sleep state, this allows the node to recharge while still maintaining some volatile state and basic clock functionality. Soft intermittency can be thought of as microcontroller duty-cycling on top of lifecycling, with the addition of an energy constraint—a node must have sufficient stored energy in order to wake up and perform a task. Thus far, most intermittent communication research has only used soft intermittency [27–30].

We have found that both hard and soft intermittency have merits, mostly depending on the relationship between the harvesting rate and the sleep power of the node [8]. To this end, Lure supports both flavors of intermittency.

2.3. Intermittent Communication

Intermittent communication (i.e., communication directly between two intermittent nodes, instead of through a third party such as an always-on node) presents a variety of challenges above and beyond the duty-cycled radio communication studied in traditional wireless sensor networks (WSNs). In traditional duty-cycling, a node can arbitrarily turn its radio on at any point in time. But in intermittent communication, a node may be unable to turn on its radio at a point in time *because the node itself may be unpowered.* Moreover, the node's on-times—when it can turn on its radio—are strongly dependent on available energy (which fluctuates over time and may be difficult to predict). In short, communication is restricted to the operating regions shown in Fig. 1, and a node has at best limited and imperfect control over when these operating regions may occur.

Thus, aligning the operating regions of two nodes such that they can communicate—i.e., achieving *overlap*—is a main challenge of intermittent communication, and that the lack of fine-grained timing control by the nodes makes this a fundamentally new problem, distinct from traditional duty-cycled radio communication. To address this unique challenge, prior work [17, 24, 25] has explored intermittency-aware techniques capable of exerting limited control over a node's lifecycle to increase the rate of *overlaps* and, thus, communication performance. From these works, an entire intermittency-aware netstack has emerged [6] that is capable of implementing intermittency-aware strategies.

Figure 2 shows the typical core components of a BISN node as presented by [6]. The power supply of a BISN consists of a harvester (e.g., [31]) that harvests ambient power and uses a capacitor for energy storage. As illustrated in Figure 1, when the capacitor voltage reaches V_{ON}, the harvester powers on the digital components of the sensor node. During operation the digital components consume energy from the capacitor until the capacitor voltage reaches V_{OFF} and the power supply shuts power off to the digital components, allowing the capacitor to charge again.[2] A sensor node typically consists of a microcontroller (e.g., TI MSP430 [34]) and a radio (e.g., TI CC1352 [35]). The power management frontend allows the software driver sitting on the sensor node to manipulate its energy harvesting and consumption pattern. This enables the node to exert limited and imperfect control over on-times using an intemittency-aware component called the Lifecycle Management Protocol (LMP) [24, 25]. The LMP, in turn, interacts with other intermittancy-aware modules to support the the network stack and to maintain a continuous and shared sense of time [17].

Maintaining a continuous sense of time is one of the unique challenges of (hard) intermittent systems. During node off-times, traditional hardware timers and real-time clocks (RTCs) are unpowered. Therefore, an intermittent sensor node is unable to keep track of a continuous sense of time. Because of this, intermittent nodes may include one or more *persistent clocks* [17, 18]. Persistant clocks measure the amount of voltage decay on small capacitors to estimate time across off-times. Readings from persistent clocks can be merged with readings from traditional "active" clocks to create a continuous sense of time.

[2] V_{OFF} is either the minimum voltage the voltage regulator needs to produce its stable output voltage or a preset value of a voltage supervisor[32, 33].

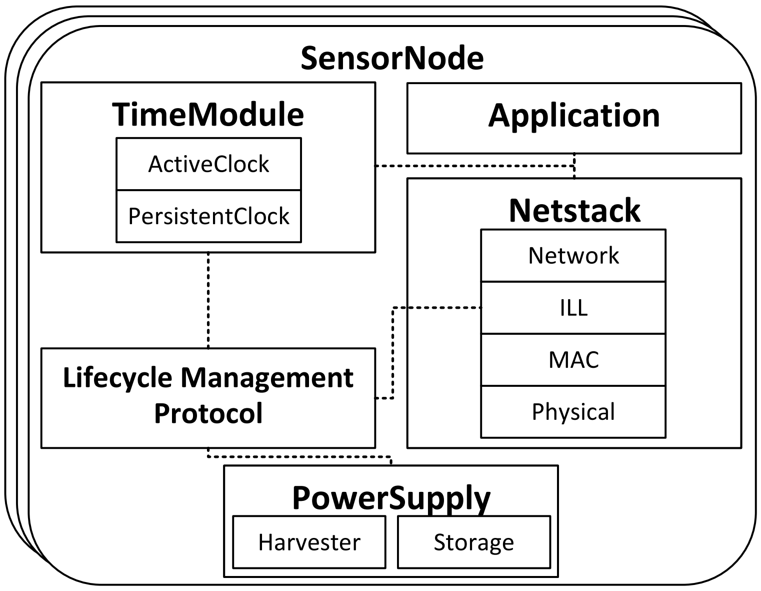


Figure 2: An intermittency-aware model for a BISN proposed in [6].

The intermittency-aware netstack overall resembles that of duty-cycled WSNs, but it includes intermittency-specific adaptations and a new intermittency-specific layer called the *Intermittent Link Layer (ILL)*. The ILL is an intermittency-specific layer responsible for managing the intermittency of the links between neighboring nodes. The ILL is also responsible for advocating for communication between other intermittency aware components in the node subsystem like LMP and time. The (co-)design and (co-)optimization of the ILL, network, MAC, LMP, and time components are the subject of active research. Relevant examples are shown in the case studies presented in Section 5.

3. A Case for a Batteryless Intermittent Sensor Network Simulator

3.1. Related Work

3.1.1. Network Simulators

A variety of network simulators have been used in prior work to evaluate related systems such as WSNs. Cooja [36] is a network simulator for sensor nodes running Contiki OS, an open-source operating system specifically designed for low-power and lossy wireless networks. Contiki includes implementations of many low-power communication standards, including RPL and 6LoWPAN. Cooja makes use of hardware emulators so that the simulated nodes run actual compiled Contiki code. This makes the transition from Contiki to deployment easier, but it also creates a sharp learning curve and slows down the prototyping process. Additionally, initial testing has shown that Contiki has a significant boot time overhead that is unsuitable for intermittent nodes.

TOSSIM is a discrete event simulator for applications written for TinyOS [37], an OS similar to Contiki. TOSSIM is written as a TinyOS library, and TOSSIM simulations may use actual TinyOS code. TOSSIM also allows individual components to be replaced with a simulated version, making it very flexible. However, a lack of activity indicates that TinyOS and TOSSIM are no longer actively developed or supported.

Another popular open-source network simulator is ns-3 [38]. It is a discrete event simulator that supports different network types such as WSNs, IoT, LTE, software-defined, Wi-Fi, distributed computing, and optical networks. ns-3 is written in C++, with a Python API. Concurrent to the development of Lure, some work has been done toward adding support to ns-3 for intermittent systems [39]. This work is focused on the hardware components and does not consider potential protocol stack changes for intermittent systems. While we acknowledge that ns-3 is powerful and fully-featured, we developed Lure in order to have a more streamlined tool specifically for intermittent systems, implementing our envisioned intermittent sensor node framework (described below) from the ground up. OMNeT++ [40] is another discrete event network simulator that has been used for WSN simulation in the past. OMNeT++ is modular and extensible, but its computer networking components were not designed to simulate energy availability [41]. Some work has been done to use OMNeT++ for intermittent network simulation [41, 42], but the general applicability of this work is unclear. Protocol-specific simulators have been developed for evaluating other work [43], but to the best of our knowledge, Lure is the first simulator specifically for simulating generic networks of intermittent nodes. Earlier versions of Lure have been used for evaluation in [8, 17, 25], and as a reference tool in the development of hardware testbeds.

3.1.2. Models for Intermittent Systems

Several works have proposed models for various components of intermittent systems. [44] proposes a model to simplify the programming of intermittent systems while ensuring that application data remains consistent across power failures. The EH model of [45] provides the ability to quantify the forward progress in an intermittent system by providing an estimate of the amount of energy used in useful program execution versus energy wasted by lost program exeuction due to node death and the checkpoint/restore process. [46] is an instruction-level simulator for MSP430 MCU-based intermittent nodes. It provides a capacitor model to simulate lifecycle behavior, support for state-of-the-art non-volatile memories such as FRAM, and peripheral support using GPIO, SPI/I2C. Additionally, it can also simulate the power draw of a CC1101 radio. [47] models complete batteryless nodes at cycle-level with detailed circuit-level models for the MCU, energy harvesters, power management circuits, energy storage, and peripherals such as sensors, radio modules. While this has very high fidelity, it does not scale to long-lasting simulations across many nodes as needed for BISN research. [48] uses high-level simulation to model timely behavior of a single intermittent system. [49] models the impact of memory technology, intermittent environment, and hardware modifications, using RISC-V ISA and Rocket Chip. All of these works [44, 47–49], however, only model computational aspects of batteryless nodes and do not model communication between them.

One line of work, energy packet networks (EPNs) [50–53], models networks of systems powered by renewable energy sources that can be intermittent in nature. These works lack modeling critical timing aspects of BISNs such as ON-time overlap, node operational OFF-times, and time synchroniziation. Other works have modeled aspects of coordinated behavior and communication between multiple intermittent nodes. [54] provides a probabilistic model for persistent clocks that maximizes the likelihood of estimated off-time values to minimize the error of persistent clocks. Additionally, it also provides a Monte Carlo simulation of network synchronization based on these probabilistic models. Although this work reduces time synchronization error between a network of BISN nodes, it does not model or enable communication using improved time synchronization. [29] enables successful communication between (soft) intermittent nodes by modeling the charging times using various distributions, such as Gaussian mixture models and online learning, to allows nodes to schedule wake up times. However, the communication stack itself is not modeled. Unlike [29], [24] performs more generic analytical modeling of communication delay and throughput between BISN nodes where each node turns itself off after a fixed, but configurable, number of communication slots—an LMP called `FixedLMP` in our simulator. This model was shown to usefully predict experimental trends when considering two nodes with static sender and receiver roles (i.e., the sender node constantly transmits while the receive node always listens) and relatively constant harvesting conditions. Due to variation in ambient harvesting power, on-time power consumption, and role changes, however, the modeling does not always accurately capture the dynamic relationship between the communication stack (e.g., IMAC and ILL) algorithms and energy harvesting contexts.

3.1.3. Detailed comparison of existing models with BISN needs and proposed simulator

Table 1 compares the state-of-the-art EPN models [50–53], state-of-the-art BISN model [24], to our proposed Lure simulator in the context of the BISN case study needs presented in this paper. While there are some shared characteristics between these works (e.g., modeling of variable/intermittent energy harvesting, packet generation, queueing, and transmission), there are several critical characteristics that are absent in state-of-the-art EPN works. Most critically, EPN works do not model nodes being completely non-operational when powered off, link-layer communication mechanisms, or time-related metrics and mechanisms.

[50] focuses on high-performance cloud servers that incorporate battery backup systems to address any interruptions in the energy supply from changes in the renewable energy sources. The comparatively large scale of the energy that can be delivered on-demand, the larger timescale over which this demand occurs, and the assumption of always being operational (i.e., powered on) makes these a fundamentally different problem

Table 1: Comparison of energy packet networks [50–53, 55] to BISN performance models and case study needs.

	Characteristic	[50]	[51]	[52]	[53]	[55]	[24]	**Lure**	**BISN Case Study**
Eval Metric	Avg compute job time	-	-	-	✓	✓	-	-	-
	Comm delay	-	✓	-	-	-	✓	✓	Sections 5.1 and 5.2
	Comm throughput	-	-	-	-	-	✓	✓	Sections 5.1 and 5.2
	Compute job throughput	-	-	-	-	✓	-	-	-
	Prob meet energy demands	✓	-	-	-	-	-	-	-
	Packet flow stability	-	-	✓	-	-	-	-	-
	Reserve energy probability	-	-	-	-	✓	-	-	-
	Time sync error	-	-	-	-	-	-	✓	Section 5.3
Application	Intermittent execution	-	-	-	-	-	✓	✓	-
	Packet/job generation model	-	✓	✓	✓	✓	✓	✓	Sections 5.1 and 5.2
Netstack	Data packet transmission	-	✓	✓	✓	✓	✓	✓	Sections 5.1 to 5.3
	Packet queueing model	-	✓	✓	✓	✓	-	✓	Sections 5.1 to 5.3
	Persistent communication state	-	-	-	-	-	-	✓	Section 5.1
	MAC level protocol	-	-	-	-	-	-	✓	Section 5.1
	Multinode ON-time overlaps	-	-	-	-	-	✓	✓	Section 5.1
	Sender vs receiver role selection	-	-	-	-	-	-	✓	Sections 5.1 and 5.2
Time	Time synchronization protocols	-	-	-	-	-	-	✓	Section 5.3
	Clock models	-	-	-	-	-	-	✓	Section 5.3
	Persistent time information	-	-	-	-	-	-	✓	Section 5.3
Lifecycle Mgmt	on-time control	-	-	-	-	-	✓	✓	Sections 5.1 to 5.3
Power Supply	Variable/intermittent energy	✓	✓	✓	✓	✓	$\sim$	✓	Sections 5.1 to 5.3
	Energy buffering/queueing	✓	✓	✓	✓	✓	$\sim$	✓	Sections 5.1 to 5.3
	On demand energy dispatch	✓	-	-	-	-	-	-	-
	Energy flow control	✓	✓	-	✓	✓	-	-	-
	Energy switches	✓	-	-	✓	✓	-	-	-
	No operation when off	-	-	-	-	-	✓	✓	Sections 5.1 to 5.3

to evaluate and optimize.

[51–53, 55] propose extending this model to wirelessly-communicating energy-harvesting sensor nodes that use capacitors or batteries to store harvested energy. However, they model the harvested energy and energy stores as sufficiently large such that nodes are always operational enough to receive or respond to packets. This assumption does not hold for BISNs as shown in Figure 1 where nodes cannot perform such functions while off, booting, or sleeping. This results in models that do not consider ON-time overlaps across nodes nor the ability to control ON-times. Additionally, these models do not model differences in the link-layer protocols and design choices which important to BISNs as articulated in the case studies in Section 5. Finally, each node's sense of time is also important for BISNs. This depends not only on clock models, but also on the ability to persist time information across off-times and to reconstruct lost time based on timing information from other nodes. Not only do EPN's lack a notion of node time, but also lack the ability to model the dependency on system components such as lifecycle management and link-layer protocols.

In summary, EPNs generally focus on optimizing task (compute or communication) completion under energy harvesting via strong control over both the use and distribution of energy. While such approaches may be useful for BISNs, the most critical challenges for BISNs managing the control of on-times, link-level overlaps, and timing under the unpredictability of intermittency. [24] represents the closest model to meeting BISN needs, but still lacks a detailed modeling of link-layer properties (e.g., sender-receiver roles) and any notion of a node's sense of time.

3.2. An Analytical Model for Intermittent Communication Performance

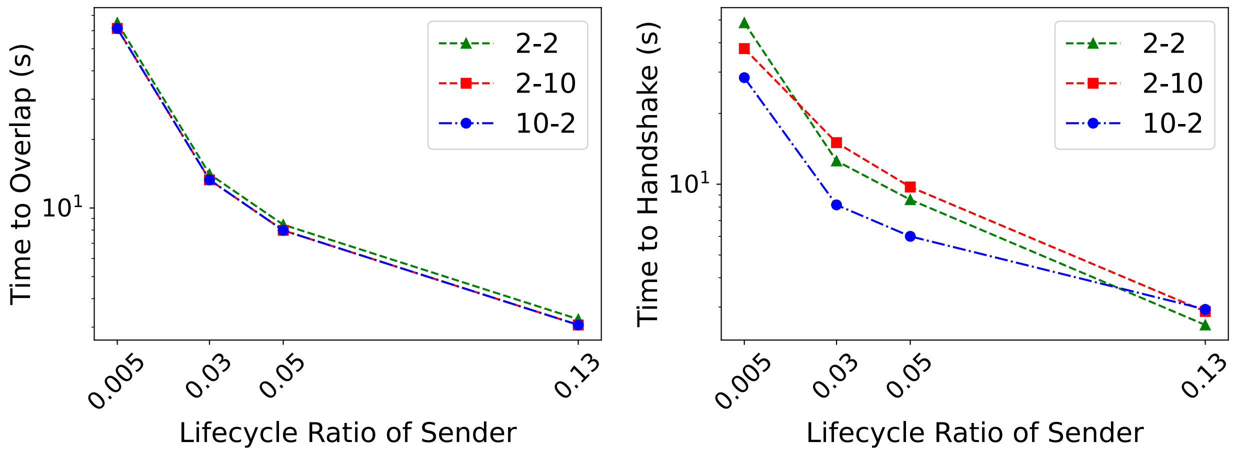


Figure 3: Comparision of analytical model (left), and real experimental data (right).

To further understand the capabilites and limitations of the best existing analytical models for intermittent communication, we further consider the analytical model for time to overlap (TTO) from [24]. TTO corresponds to an experimental metric called time to handshake (TTH). TTO and TTH are essentially measures of communication latency between two nodes when the sender is saturated with packets to deliver to the receiver. The model in [24] assumes *fixed values for* the mean harvesting power of each node, on-time power consumption, boot time, and maximum on-time.

In Figure 3, we compare the TTO of the analytical model with TTH data obtained from experiments on a testbed (see Section 4.4 for testbed details). The average lifecycle ratio of the sender node varies from 0.005 to 0.13 on the x-axis while the average lifecycle ratio of the receiver node is around 0.1 in all scenarios. We consider three different high-performing configurations of `FixedLMP` for the two nodes: 2-2, 2-10, and 10-2, where the first number is the default on-time slots of the sender and the second number is the default on-time slots of the receiver.

Overall, the analytical model captures the general lifecycle ratio trends of the experimental data. For example, TTO and TTH decrease in a similar manner as the lifecycle ratio increases from a very low value of 0.005 to a higher value of 0.13. However, the analytical model fails to capture the relative differences of TTH between 10-2, 2-10 and 2-2 configurations as observed in the experimental data. For example, at a lifecycle ratio of 0.03, the experimentally measured TTH when the sender is on for 10 slots (i.e., 10-2) is at least 35% lower than when it is on for only two slots (i.e., 2-2 and 2-10). In comparison, the analytic model indicates that 10-2 and 2-10 are identical in terms of TTO and within 5.6% of 2-2. Essentially, the analytic model cannot distinguish between the roles that each node takes in communication, which can lead to an incorrect choice of LMP when designing the higher-level communication stack components such as the ILL. Furthermore, this model would perform worse under less ideal conditions (such as non-Gaussian distributions of harvesting power), and this model only works for one specific and relatively simple communication protocol.

3.3. A Case for Simulation

Evaluating novel solutions for WSNs has always been challenging. As discussed in Section 3.1, analytical modeling is generally insufficent in cross-stack BISN performance evaluation while hardware testbeds with custom prototype nodes require time and effort to build and maintain. Embedded nodes can be difficult to instrument, and achieving experimental repeatability is hard. Large-scale deployments often are not feasible for researchers, limiting the scenarios that can be explored in testbed experiments. Additionally, in order to achieve statistically significant results in WSNs, experiments may need to run for a long time (e.g., because packets may be sent infrequently). Simulations can often be performed faster and at lower cost than testbed experiments, and multiple simulations can be run in parallel. Thus, simulation has become an important component for evaluation of WSN research.

These challenges of traditional WSN evaluation are exacerbated in BISNs, particularly in terms of repeatability and experiment duration. Since batteryless nodes rely on energy harvesting, the repeatability of an experiment depends on repeatability of the ambient conditions, which may generally be out of the control of the experimenters. This may be solved with an energy replay device [56], but this adds cost and complexity to the testbed, and synchronized replay for networks has not yet been developed. Moreover, compared to a traditional WSN experiment, a BISN experiment's duration likely needs to be multiplied by a factor dependent on the LCRs of the nodes to achieve the same level of statistical significance. In practice, this factor may be very large; for example, due to the challenges of intermittent communication, packets may only be successfully sent during a small fraction of the nodes' on-times (e.g., the model in [24] predicts overlap in about 1% of on-times for a baseline protocol with an LCR of 0.1). A BISN simulator is therefore crucial for enabling rapid prototyping and repeatable, statistically significant experimentation.

4. Evaluation

In this section, we first provide a brief description of Lure, our software simulator for BISNs, and discuss calibration of Lure in terms of the energy models used to drive the intermittent behavior of the simulated nodes. We then present a validation of Lure in the form of behavioral traces and simulation results compared to experimental results from a testbed of prototype physical nodes powered solely by energy harvesting and running a software framework that mirrors Lure's `SensorNode`. The purpose of this validation is to show that Lure works as expected and generates results that follow the trends of reality—i.e., Lure sufficiently captures the nuances of intermittent behavior to make it a useful tool for evaluating intermittent protocols and optimizing intermittent network performance.

4.1. Lure Simulator

Lure is a discrete event simulator designed as a research tool for efficiently exploring batteryless intermittent communications. It is implemented using object-oriented Python with a JSON (JavaScript Object Notation) configuration system to provide both modularity and configurability, as described in detail in Appendix A.

Lure uses an EnergyModel object to decide the instantaneous harvesting power and a SensorNode object to model each BISN node. Every SensorNode object has a (1) power supply, (2) a lifecycle management protocol (LMP) module, (3) an intermittent time module, (4) a traffic generator, and (5) an intermittency-aware network stack (netstack) component as described in detail in Appendix A. Additionally, Lure contains `Logging`, `Stats` and `Plotting` modules for data collection, processing, and analysis. All simulator source code, case study scripts, and results are available at `https://zenodo.org/records/11062224`.

4.2. Energy Model Calibration

The energy available to a BISN is a primary factor in determining its intermittent behavior, so the accuracy of the energy model is critical for Lure. While the available energy can be specifically modeled to match a particular environment, energy source, and application, we intend to provide a reasonable set of defaults in Lure. To this end, we ran a series of hardware experiments measuring off-time durations for (hard) intermittent nodes over a period of time. Since the off-time is dictated by the charging rate, this off-time data serves as a proxy for the available energy in the environment. The hardware was an MSP430FR5994 [57] device connected via SPI to a CC1352R [35] radio device, harvesting from a Powercast P2110 [58] RF harvesting device. We ran these experiments with various fixed durations for the on-times of the nodes (analogous to `FixedLMP` configurations) in a lab setting with typical amounts of RF noise (WiFi, Bluetooth, microwave, etc.) over a number of weeks.

Our main takeaway from these experiments is that it is reasonable to model available RF energy using Gaussian distributions. For example, Fig. 4 shows a histogram of off-time durations when a node remains on for one 5 ms slot (after booting) each lifecycle. In this experiment, the data reasonably fits a single Gaussian distribution. In other experiments, temporary perturbations in the system caused a shift in off-times for part of the experiment; most of these cases, such as that shown in Fig. 5, fit a Gaussian mixture model fairly well. Based on these observations and the fact that they align with prior work (e.g. [29]), we choose to provide a Gaussian available energy model as the default in Lure, while noting that Lure's `EnergyModel` can easily be subclassed to support more complex models or trace-based input. We use the Gaussian model for the remaining evaluations.

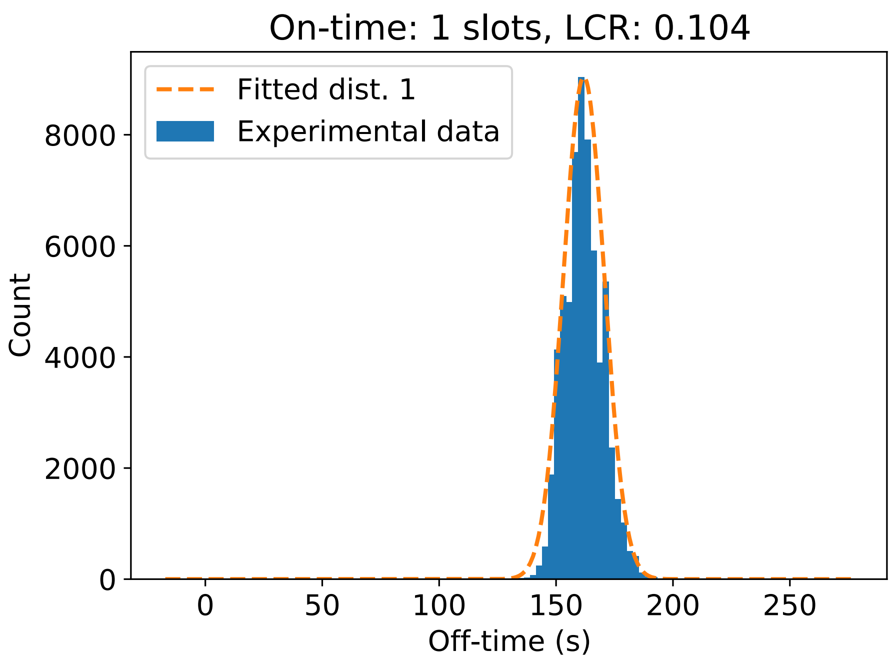


Figure 4: An experimental histogram of off-times and a single fitted Gaussian distribution.

The next question is what parameters to use for the Gaussian distribution. The mean represents the strength of the ambient energy source. For convenience, we select a default mean for the `EnergyModel` that roughly corresponds to the operating power of a typical node. This allows the `Harvester` of each node to apply a simple scaling factor to the available energy in order to achieve the desired LCR for that node. To find a default standard deviation, we filter the off-time duration dataset to only include the main Gaussian distribution from each experiment. We plot the standard deviations of these distributions vs. the means in Fig. 6. The standard deviation appears to generally increase with the mean; we plot a linear regression line to illustrate this. From this data, we determine that a standard deviation of approximately 4% of the mean is reasonable as the default.

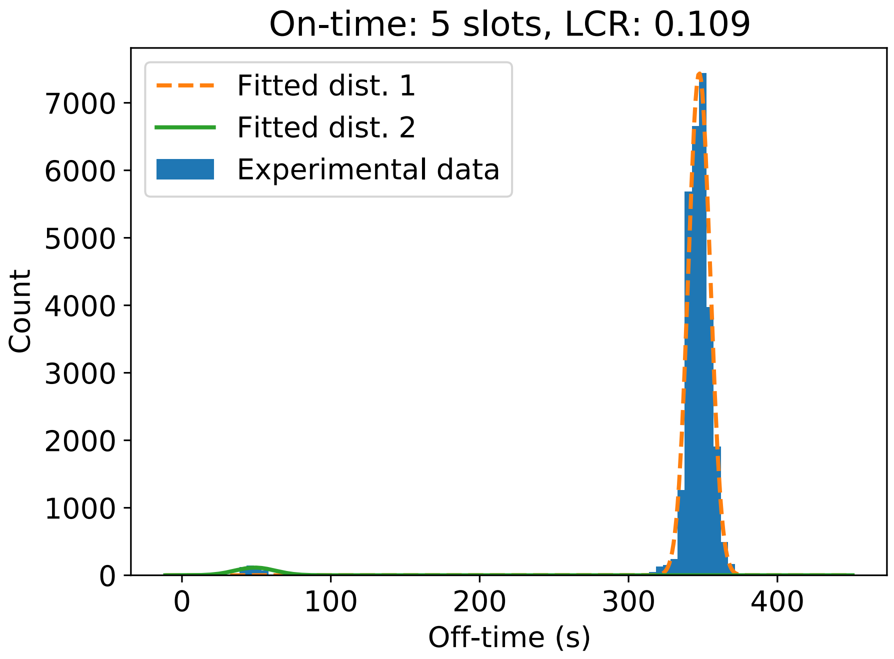


Figure 5: Another experimental histogram of off-times and a bimodal fitted Gaussian distribution.

4.3. Intermittent Behavior Validation

Lure is purpose-built to support intermittency. This means that it simulates intermittent behaviors with a high level of detail and precision. To validate and demonstrate this, we first present an experimental trace of intermittent operation for a single node in Fig. 7. This trace was captured from a logic analyzer attached to a hardware node similar to those used in the off-time experiments above. The trace clearly shows node lifecycling—turning on, booting, and then turning off (in this case, because of the LMP configuration). The top axis shows when the node is transmitting at the MAC layer. Using our default MAC protocol (mirroring that implemented in Lure), the node attempts three transmissions before its LMP turns it off.

For comparison, we used Lure's built-in time series plotting function to plot traces of selected simulation state values for a single on-time in Fig. 8. As in the physical hardware trace, the node turns on, boots, and transmits three packets before its LMP turns it off. Overall, the traces exhibit a high degree of similarity. In particular, note the accuracy with which Lure models the voltage across the energy storage capacitor. This voltage level is key to a solid foundation for intermittent node simulation (though Lure does not currently simulate the transients shown in the logic analyzer trace, as they do not qualitatively affect the operation of the node).

Next, we present a logic analyzer trace (Fig. 9) and a Lure trace (Fig. 10) showing two nodes lifecycling, overlapping, and communicating. The overall similarity of these traces further confirms that Lure's simulation of node lifecycling behaviors, as well as when the nodes are and are not able to communicate, is accurate.

We also emphasize that instrumenting, visualizing, and debugging intermittent node behavior is much easier in Lure than on physical hardware, which is a key motivation for having a simulator for these types of systems. Our own experience has even shown that insights from Lure can be used to debug issues observed on physical hardware.

4.4. Intermittent Communication Validation

Having validated Lure's intermittent behavior at the node level, we now evaluate Lure's ability to model intermittent communication. Using the same physical hardware as in the previous section, we run a series of two-node experiments where one node continuously attempts to send packets to the other node. Both nodes are (hard) intermittent, with the receiver initially set up to average an LCR of 10% and the sender's

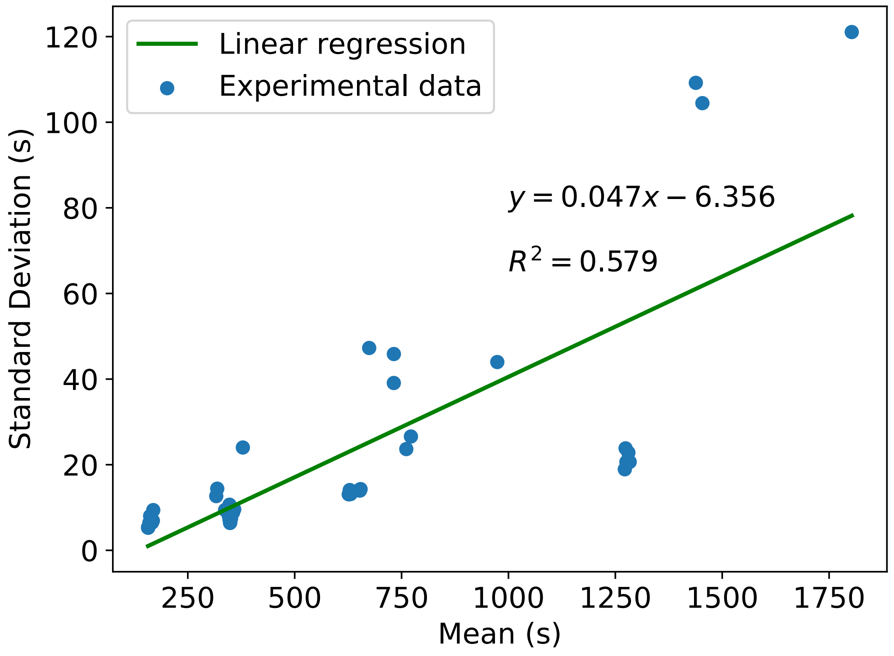


Figure 6: Standard deviation vs. the mean of the filtered experimental off-time data.

LCR varying according to the experiment. For software, the nodes use an implementation of the `FixedLMP` and `BasicILL` protocols described above for Lure, as well as a similar MAC protocol. We evaluated four different configurations of `FixedLMP` for the two nodes: 2-2, 2-10, 10-10, and 10-2, where the first number is the default on-time slots of the sender, and the second is the default on-time slots of the receiver. The communication slot time is measured to be around 12 ms, mostly due to software and SPI delays. We also evaluate Null-LMP, where the nodes simply stay on until they run out of energy (approximately equivalent to 12-12 with the 12 ms slot size).

We duplicate this setup in Lure and present results for both simulated and experimental communication performance in Fig. 11. The metrics we present are throughput (packets successfully received and acknowledged per second) and time to handshake (TTH), an intermittent-specific metric that measures the time between successive overlaps in which communication occurs (with potentially multiple packets sent per overlap). Each simulated datapoint is the average of 40 trials, with 1000 packets sent in each. The experimental results are for 4–12 (depending on the LCR) trials of 1000 packets each. The shaded areas indicate 95% confidence intervals.

Comparing the simulated and experimental results, our first observation is that both TTH and throughput results are reasonably close in an absolute sense. We believe that with additional experiments, Lure could be further calibrated to make the absolute results closer. Much of the difference can likely be explained by the small experimental sample size, experimental edge cases that have not been debugged, small differences in timing and power parameters, and external events and perturbations such as LCR drift. However, setting up, running, and debugging these experiments is time-consuming (motivating Lure's existence), so we leave additional calibration to future work. More importantly, we observe that Lure already generally mirrors the performance *trends* seen in the experiments. Experimental confidence intervals notwithstanding, the shapes of the TTH and throughput curves are very similar. The clusters of the LMP configurations also match, with 2-2, 2-10, and 10-2 performing significantly better than 10-10 and Null. In some cases, the specific relationships between these configurations are mostly intact—for example, 10-2 shows the best TTH performance on average for most of the LCRs in both simulated and experimental results. In other cases, these finer points are lost within the experimental confidence intervals. In general, though, we consider these results to be strong evidence that Lure can help a researcher accurately assess proposed protocols relative to each other.

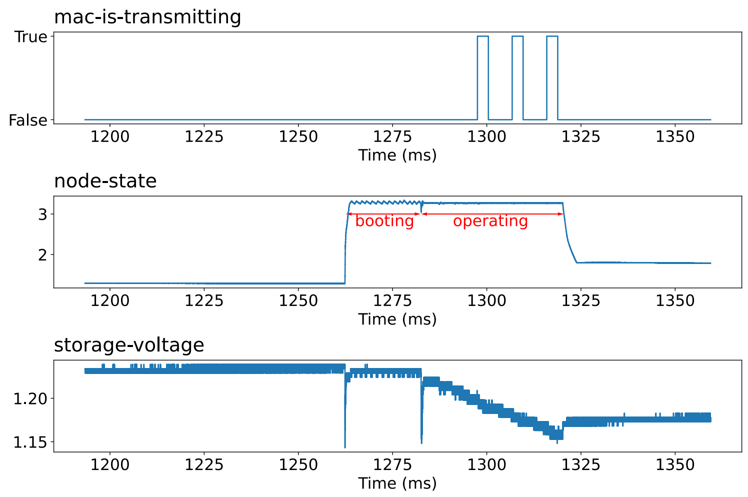


Figure 7: Plot of a logic analyzer data capture illustrating a single on-time of a batteryless intermittent node in our experimental testbed.

4.5. Scalability of Lure

To support rapid prototyping, simulations with a reasonable number of nodes should complete much faster than experiments on physical hardware. To assess Lure's simulation speed and scalability, we measured the runtime of a series of simulations. Our goal is to compare real and simulated time as the number of nodes in an experiment increases. We run our experiments on a 64-core Intel Xeon Gold 6130 2.10 GHz system with 500 GiB of RAM. We configure the nodes to use `FixedLMP` with two communication slots of on-time. We set the round-trip communication slot length to 5 ms[3], for a total of 10 ms of on-time by default. We use `BasicILL`, which extends the on-time of the LMP when communication occurs. Logging during the simulations is limited to a single entry per simulated minute to track the current real-time duration of the simulation. Data collection is restricted to scalar statistics such as the number of packets generated and total simulated on-time. We use the default Gaussian energy model with a 10 s update interval.

We arrange the nodes in a grid pattern with sufficient communication range to communicate with adjacent nodes, including on the diagonal, resulting in physical topologies such as that shown in Fig. 12a. Node 0 is the designated sink node and generates no packets. All other nodes generate traffic using a Poisson distribution at a rate of one event per second. All packet-generating nodes are configured with a static route to Node 0 that results in a network topology such as that shown in Fig. 12b.

We conduct experiments with an increasing number of nodes to evaluate how Lure performs as the scale of simulations expands from 2 to 225 nodes. The simulated duration of each simulation trial is one hour. For each network size, we conduct 10 trials for a total of 10 hours of simulated time. To showcase Lure's multiprocessing capability, each experiment is conducted using 1, 2, 5, and 10 processes. In each case, the real time to complete the experiment is measured, and the results are shown in Fig. 13.

The results of these experiments show two important benefits of Lure. First, under these conditions, single-process Lure simulations have a real-time duration that is equal to or faster than simulation time when evaluating up to around 100 nodes, demonstrating Lure's scalability in terms of runtime. Second, Lure's multiprocessing capabilities allow it to simulate many trials in parallel, which significantly increases the speed of even large-scale simulations. While multiprocessing may involve increased overhead, its benefit is clearly seen. Indeed, executing the suite of experiments with a single process took Lure almost 34.6 hours to complete, whereas using 10 processes took just over 4.3 hours to finish.

Additionally, we collected virtual memory consumption for these experiments. Lure utilized a maximum of approximately 0.911 GiB of memory during the single-process experiment and 10.37 GiB during the 10-

[3]In the previous section, we used a slot length of 12 ms. Both numbers were experimentally measured for different iterations of the experimental code; the key takeaway here is that Lure supports any communication slot length.

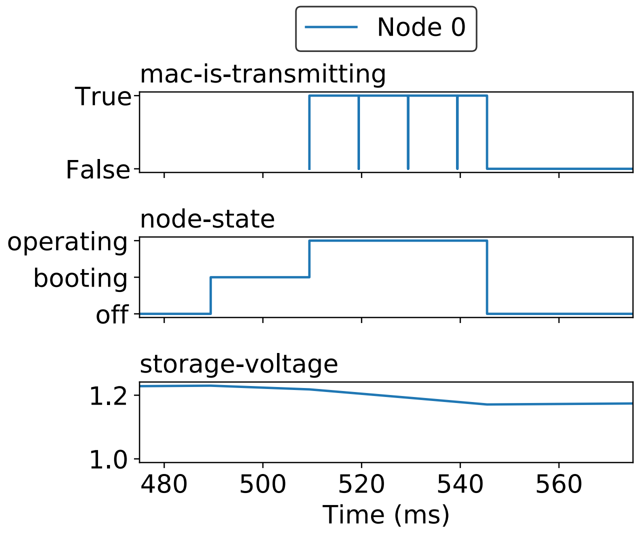


Figure 8: A trace of selected simulation state values for a single on-time of a batteryless intermittent node in Lure.

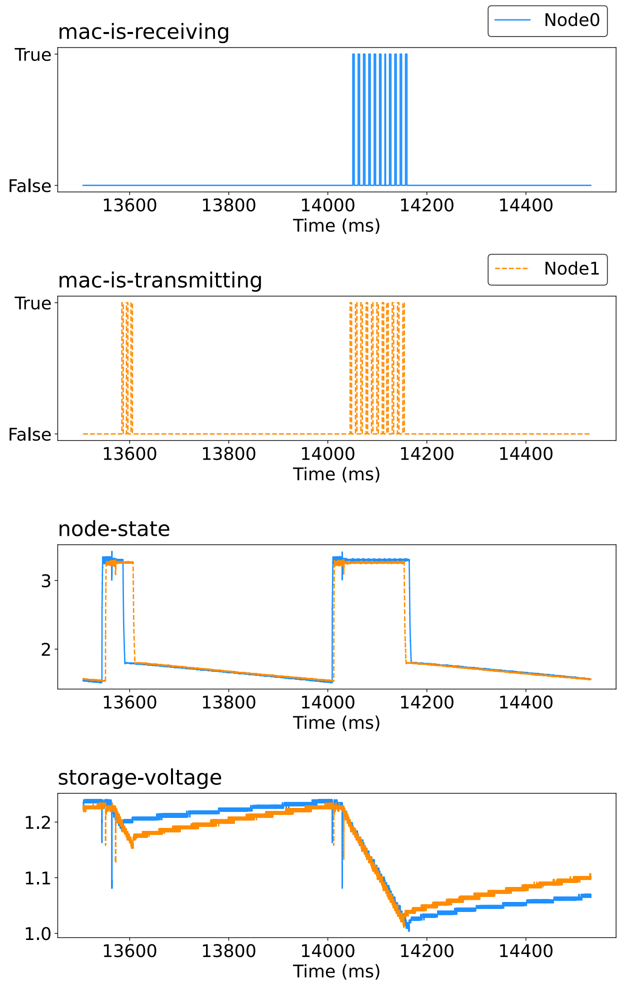


Figure 9: Plot of a logic analyzer trace of two intermittent nodes communicating with each other.

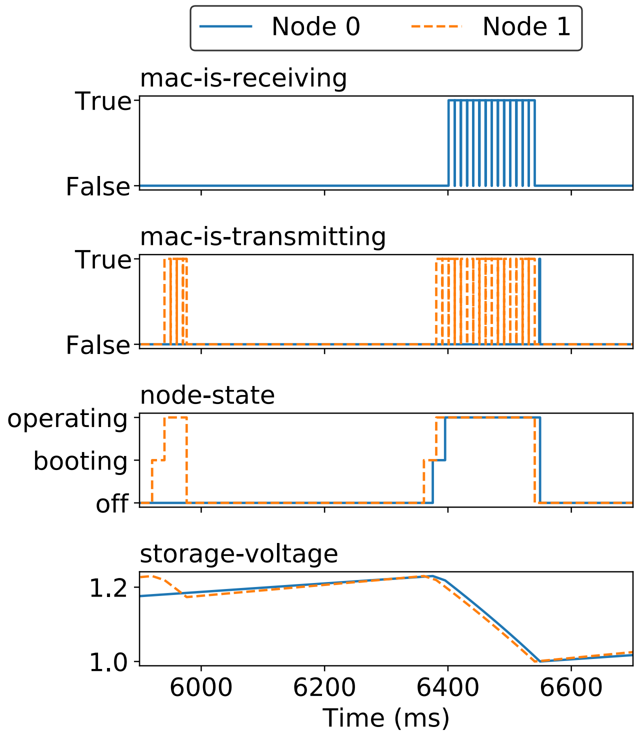


Figure 10: A trace of selected simulation state values for a period of simulated time that includes overlapping on-times and communication between the nodes.

processes experiment (1.037 GiB per process). The amount of memory increases as more time-series statistics are collected, but these numbers show that, as a baseline, Lure's memory consumption is scalable and small enough for Lure to run on a wide variety of hardware.

5. Case Studies

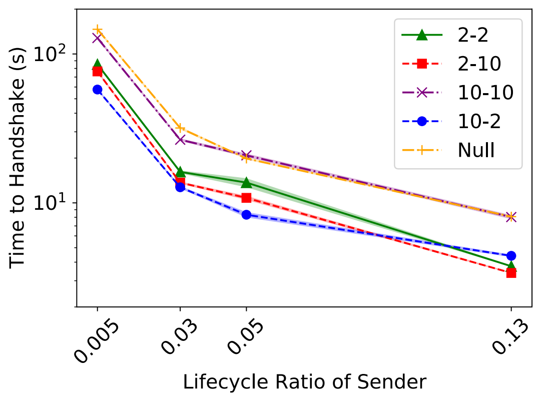
In this section, we present a series of case studies that show the use of Lure to explore various BISN-related research questions. The purpose of these case studies is not necessarily to present novel BISN research, but rather to paint a broad picture of the types of research questions Lure is designed to answer, the process for using Lure in answering them, and the types of results Lure is capable of producing. The materials for reproducing the results of each of these case studies are included as examples in Lure's source code.

5.1. Intermittent Link Layer Protocol

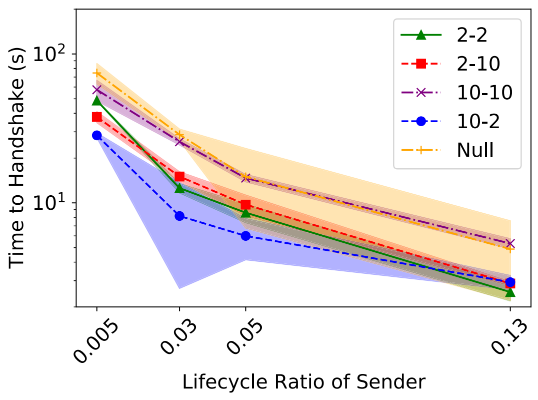
As described in Appendix A, the ILL is an intermittency-aware netstack layer, largely responsible for interfacing the netstack with other intermittency-aware components of a node, such as the LMP. As the ILL is an emerging netstack layer, design of novel protocols to manage the intermittency of links is a rich research area. Lure has been designed from the ground up to enable such research.

5.1.1. Research Question

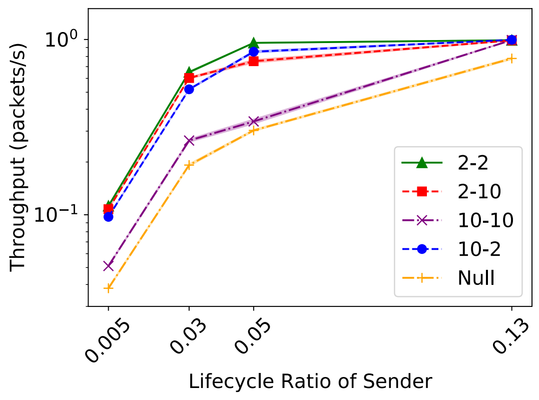
In [24], we modeled and experimented with a pair of hard intermittent nodes communicating on top of a static LMP similar to Lure's `FixedLMP`. We observed that the nodes achieved the highest throughput and lowest delay when using an asymmetric LMP configuration—e.g., when one node was configured with an on-time of 20 communication slots, and the other configured with an on-time of two slots. This observation naturally leads to the idea of a protocol where two nodes that want to communicate try to use asymmetric LMP configurations. The research question of this case study is whether and how such a protocol may improve the communication performance between hard intermittent nodes in practice.



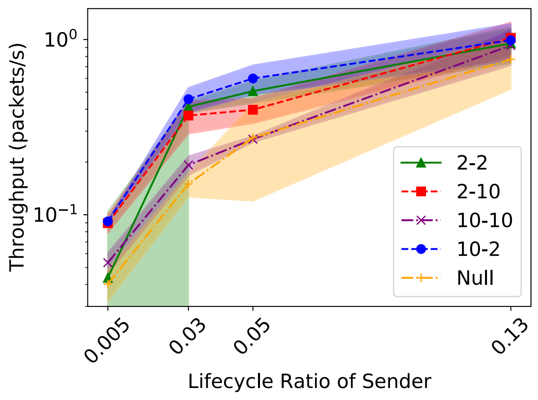
(a) TTH vs. LCR results from Lure.



(b) TTH vs. LCR results from experiments.



(c) Throughput vs. LCR results from Lure.



(d) Throughput vs. LCR results from experiments.

Figure 11: Experimental validation results. Series are labeled with the LMP configurations for the sender and the receiver, e.g. 2-10 means the sender is on for two communication slots by default and the receiver is on for 10 communication slots by default. "Null" means that the nodes stay on until they run out of energy. The shaded areas indicate 95% confidence intervals.

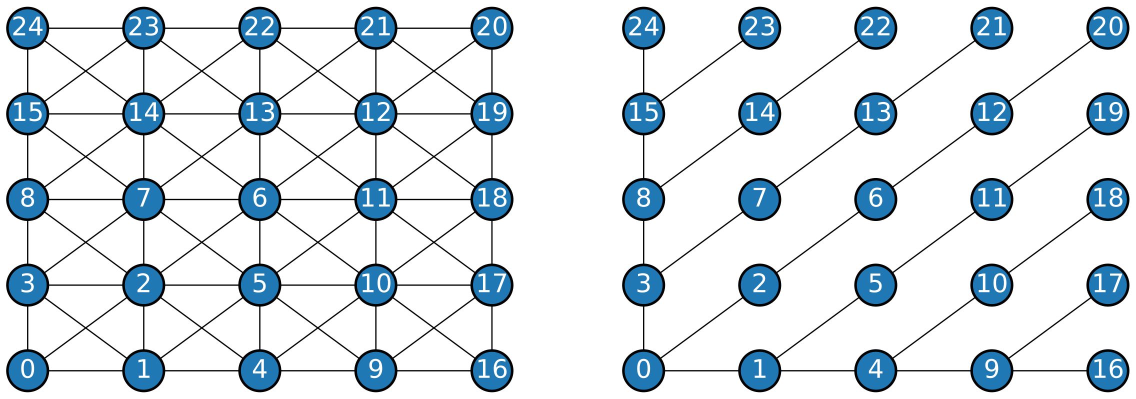
5.1.2. Implementation

We study a simple example of such a protocol in which sending node use a particular LMP configuration, while idle nodes (receivers) use a different LMP configuration. We name this protocol `GILL` (as in "Good-enough" ILL) and implement it as a subclass of `BasicILL`. We choose `FixedLMP` as the `LMP`, and we override `set_config` and `get_config` in `GILL` to add two new configuration options: `RECEIVER_ON_SLOTS` and `SENDER_ON_SLOTS`.

In `GILL`, we override `boot` to reconfigure the `LMP` to have an on-time of `SENDER_ON_SLOTS` if there are any packets in the persistent queue, and `RECEIVER_ON_SLOTS` if not. This configuration is done by calling `set_config` on the `LMP`. We also override `send_packet` to (after calling the superclass functionality) put the `LMP` in sender configuration if it is not already. Finally, we override `packet_sent` to check if the queue is empty after sending each packet, and if so, put the `LMP` in receiver configuration. This is all the development needed to implement our protocol in Lure, which is fewer than 100 lines of code in a single file. We also add a single line of code to a Python-specific `__init.py__` file in order to "register" `GILL` as a usable module in Lure.

5.1.3. Evaluation

Having implemented our new `ILL`, we configure experiments in JSON to evaluate it. We set up two-node experiments where both nodes use `GILL` and both nodes generate traffic for the other using the



(a) Physical topology.

(b) Network topology.

Figure 12: A 25-node example of the physical and network topologies used in the scalability studies.

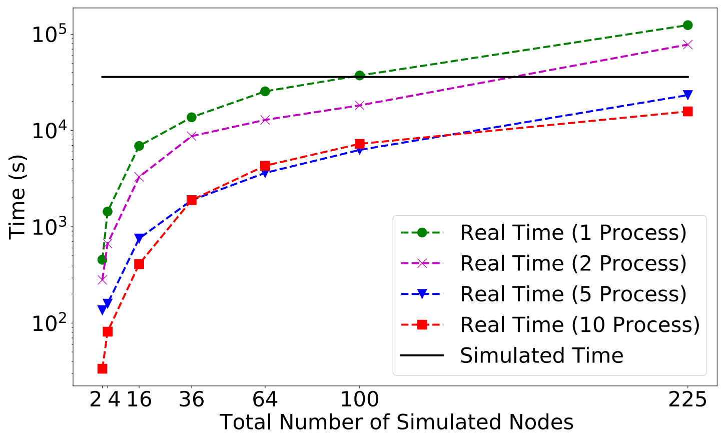
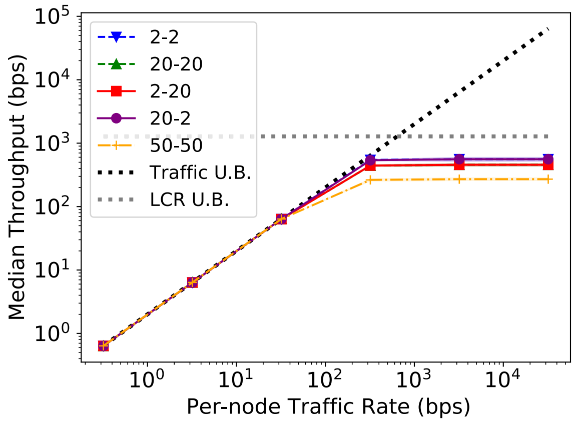


Figure 13: Elapsed real time versus the number of simulated nodes. To provide context, the horizontal line shows the total amount of simulated time.

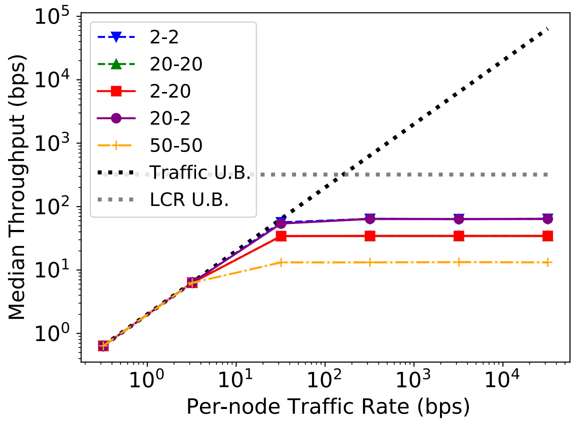
`PoissonTrafficGenerator`. Based on our results from [24], we configure the communication slot length (in the `Netstack`) to be 5 ms, and we define five `DataSeries` for the experiments: 2-2, 2-20, 20-2, 20-20, and 50-50, where the first number is `RECEIVER_ON_SLOTS` and the second is `SENDER_ON_SLOTS`. We desire to evaluate each of these configurations across a range of traffic rates, so we configure the `rate` parameter of the `PoissonTrafficGenerator` as a list. Finally, we configure this to run twice, as two `Experiments`, one with an LCR of 0.05, and one with an LCR of 0.2.

Results for these experiments are shown in Fig. 14. These are plotted using Lure's built-in `Plotter` class. The plots show median throughput and median delay versus the traffic rate for the two configured LCRs. The throughput plots also show two upper bounds: a traffic upper bound, which is the theoretical throughput if all generated packets are successfully sent, and an LCR upper bound, which is the theoretical throughput if all on-time of the nodes is spent successfully communicating.

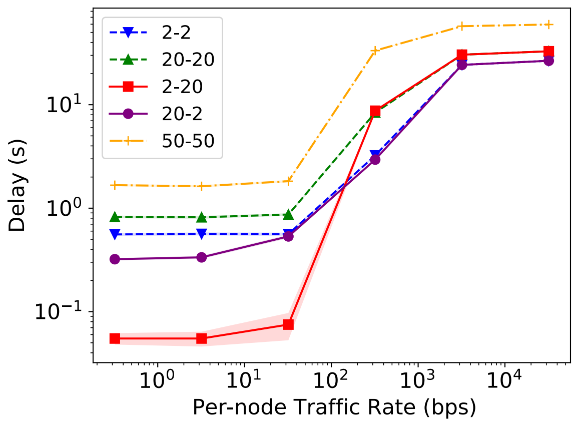
The throughput results are fairly intuitive. At a traffic rate lower than the LCR upper bound, the system is able to send all generated packets and reach the traffic upper bound. On the other hand, at a traffic rate higher than the LCR upper bound, the system is saturated, the send queues of the nodes are always full, and the traffic rate has no noticeable effect on the throughput; instead, the throughput is limited by the available energy (LCR). The gap between the LCR upper bound and the actual throughput represents the wasted on-time, and this gap could be shrunk with novel intermittency-aware protocols.



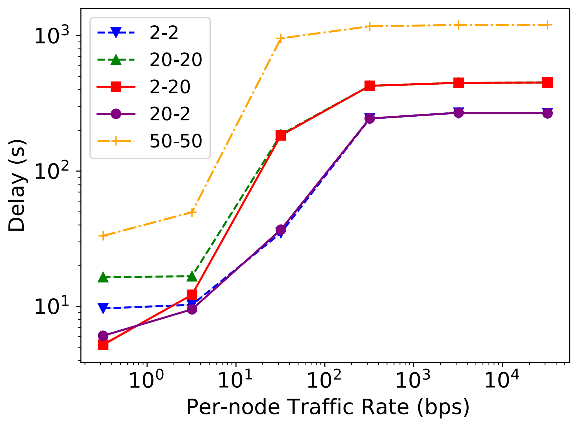
(a) Throughput w/ LCR of 0.2.



(b) Throughput w/ LCR of 0.05.



(c) Delay with LCR of 0.2.



(d) Delay with LCR of 0.05

Figure 14: Throughput and delay versus traffic rate.

Interestingly, there is no noticeable throughput benefit of using asymmetric LMP configurations. This is because at lower traffic rates, the traffic upper bound is reached for all configurations, while at higher traffic rates, there is nearly always a packet in the send queue. In this case, both nodes nearly always have their LMP configured with `SENDER_ON_SLOTS`. This explains why 2-20 matches 20-20, and why 20-2 matches 2-2. 50-50, a configuration in which nodes always take their maximum on-time, performs the worst, as predicted and demonstrated in [24].

The delay results, on the other hand, do show a separation between symmetric and asymmetric configurations. At very low traffic rates (a range reasonably likely to be targeted by an intermittent system), the nodes are more likely to be listening idly with the LMP configured to `RECEIVER_ON_SLOTS`. Due to this, the asymmetric configurations perform significantly better. In particular, 2-20 has an order of magnitude lower delay than 20-20 at low traffic rates with an LCR of 0.2—but the delays of these two configurations are identical at higher traffic rates, for the same reason as for throughput. The separation between symmetric and asymmetric LMP configurations is less pronounced for the LCR of 0.05, because with the lower LCR, the traffic saturation threshold is lower.

In summary, in this case study, we have used Lure to implement and evaluate a simple intermittency-aware communication protocol. The ease with which this was accomplished confirms Lure's usefulness as a research tool, and we plan to perform many similar but more in-depth studies in the future.

5.1.4. Comparision of Lure Simulator to Analytic Model

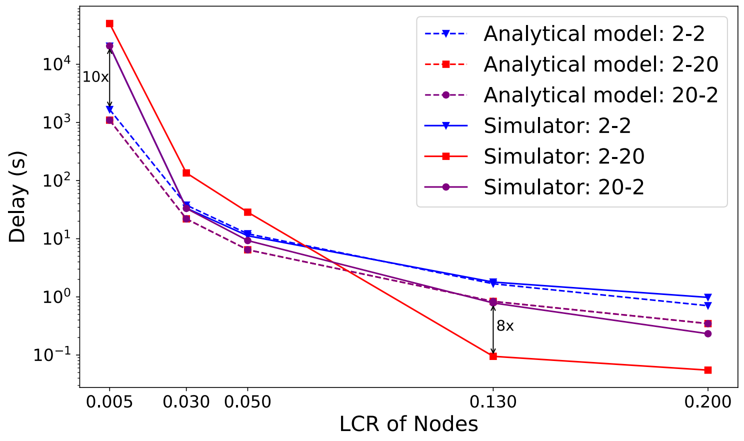


Figure 15: Delay comparision of simulator vs analytical model for different lifecycle ratio's for low traffic rate of 3.2 (bps).

Here we compare Lure with the analytical model described in [24] to answer the research question posed in Section 5.1.1. We investigate packet delay in a low traffic rate of 3.2 (bps) scenario from Fig. 14 as reasonable target for intermittent systems. As described in Section 3.1.3, the model in [24] allows us to model the delay using time to overlap (TTO). To replicate the evaluation described in Section 5.1.2, we model the delay between packet arrival to successful delivery (i.e., the next overlap). The expected event inter-arrival time is calculated based on a Poisson point process with the specified traffic generation rate parameter. Since the analytic model [24] does not model the queue, it has a default queue size of 1 — i.e., the model ignores the impact of a second packet arriving within the expected value of a TTO.

As shown in Figure 15, the analytical model can faithfully model delay for symmetric receiver-sender configuration (i.e., 2-2) compared to the simulator for LCRs greater than 0.03. At 0.005 the analytic model shows lower delay than the simulator since it is LCR bound rather than traffic rate bound and a non-negligible number of packets arrive between overlaps. In Lure (and our experimental prototype), any packets that cannot be delivered during an overlap remain in the queue, increasing their delay. Therefore, the 10x difference can be accounted for by the lack of a queueing model. While the analytic model captures symmetric ILL configurations, it fails to accurately capture delay performance for assymetric ILL configurations (e.g., 2-20 and 20-2). For example, at an LCR of 0.13, the simulator indicates that 2-20 has 8x less delay than 20-2. Using the analytical model, however, the delay remains same. The inability of analytical model to account for sender-receiver roles in ILL configurations means that it cannot be used to select an appropriate ILL protocol unlike Lure.

5.2. Network Layer

The prior case study focuses on two-node communication. In this case study, we demonstrate Lure's ability to support multinode and multihop communication, with a focus on configuring and evaluating the `Network` and `Physical` layers of Lure's `Netstack`.

5.2.1. Research Question

In traditional WSNs, fairness can be defined as the ability of each node to receive an equal number of packets from every other node in the network [59]. As traffic through a node increases, it becomes unable to handle all the packets it is being asked to forward. The introduction of intermittency to a network further complicates fairness in a network. The low energy availability that characterizes BISNs may exacerbate the congestion of a network due to a node having fewer opportunities to send and receive packets. Energy availability can also vary from node to node based upon the type of harvesting and ambient conditions in a BISN. With this in mind, BISNs pose a unique challenge to fair communication due to the added complexity of intermittency.

The research question of this case study is to explore the impact of asymmetric energy availability, relative to a node's neighbors, on fairness in a BISN. To evaluate this, we choose a six-node ring topology for both the physical and network topologies, because it requires multi-hop packet delivery and is straightforward to implement and understand.

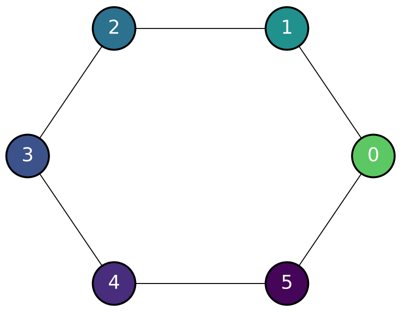


Figure 16: The topology used for the network case study.

5.2.2. Implementation

The foundation for this case study is already part of Lure; only configuration files need to be created to execute this experiment. Specifically, the physical topology, network routes, and data collection need to be configured by overriding default values for each node in the topology.

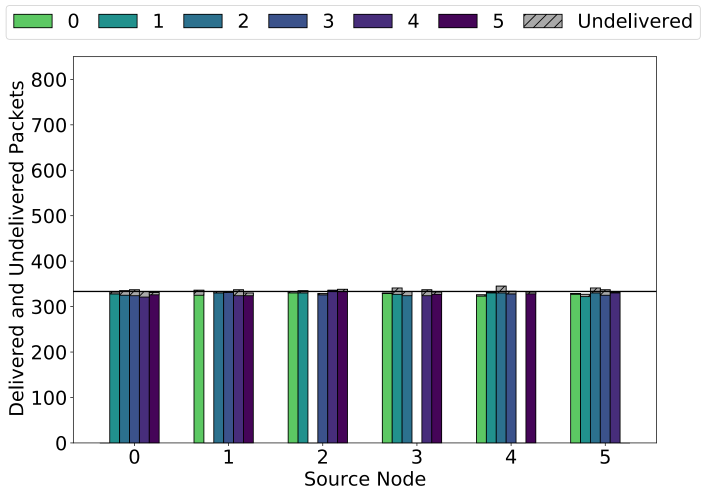
To implement the physical topology, the `ManualPhysical` class is used to configure each node with a list of other nodes in its neighborhood. In order to establish the network topology, a routing table is specified for each node using the `StaticNetwork` class. The routing tables are configured to take the shortest number of hops to each node, with counter-clockwise hops (from Fig. 16) used in the event of a tie. All nodes are configured to use `FixedLMP` and maintain a packet queue with a maximum size of 25 packets for each node in its neighborhood. All nodes generate packets and have an equal chance to choose any of the other five nodes in the system as the end destination for each packet. A Poission point process is used to control packet generation and is configured with a rate of one packet/s. All experiments are conducted until 10,000 packets have successfully reached their final destination. To capture the necessary information for this case study, two statistics are recorded: one to track the delivery of packets to their final destination, and one to count the number of packets a node generates for each destination. The former is available in Lure by default, and we implement the latter for this experiment. Lure already tracks the number of packets a node generates as a scalar stat; here, we modify this stat to be a list containing the destination address of every generated packet.

5.2.3. Evaluation

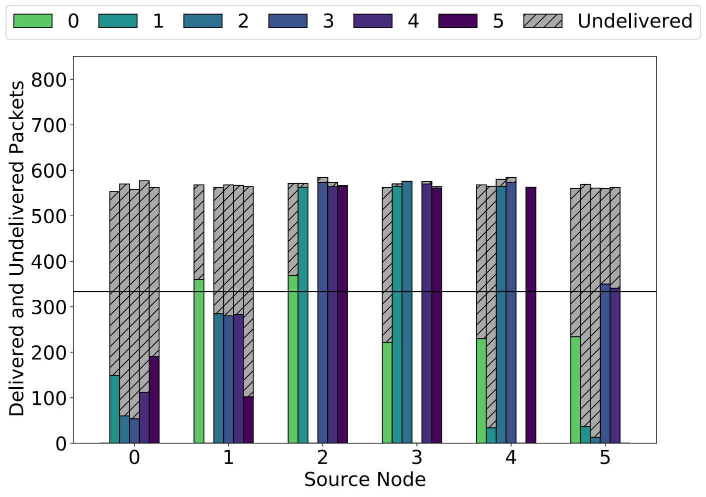
We conduct two experiments to evaluate the impact of energy availability by modifying the LCR of nodes in the ring topology. In the first experiment, all nodes are assigned an LCR of 0.2, and in the second experiment, Node 0 is assigned an LCR of 0.02 while all other nodes remain at 0.2.

In Fig. 17, the packet delivery ratios are shown. Each grouping of bars represents all packets generated and delivered by a particular source node. Each individual stacked bar represents the packets that were successfully delivered to their end destination (colored) and the excess packets that were generated for that destination but undelivered (grey and textured). The results are averaged across 20 trials. Both figures have a threshold line at 333 packets to represent the height each bar would be in a perfectly fair system.

In Fig. 17a, all nodes are assigned an LCR of 0.2 and each source node sends a similar amount of packets to each of its possible destinations. There are also very few packets that are generated but unsent. This shows that, with these simulation parameters, the system is able to maintain and distribute packets fairly. However, when a single node has its LCR reduced, as in Fig. 17b, fairness in the system falters. With an LCR of 0.02, Node 0 has trouble sending and receiving packets relative to its neighbors. This can be seen in Node 0's lower delivery rate across all destinations when compared to Fig. 17a and the threshold line. The impact of a lower LCR does not just affect Node 0. Any communication using Node 0 as a forwarder is also



(a) All nodes have an LCR of 0.2.



(b) Node 0 has an LCR of 0.02 while all other nodes have an LCR of 0.2.

Figure 17: Packet delivery and generation by each source node. The threshold line represents the height all colored bars would be in a perfectly fair system.

affected, which includes packets from 1-to-5, 4-to-1, 5-to-1, and 5-to-2. Each of the bars corresponding to these source-to-destination pairs is significantly lower than the threshold line.

These results show that the intermittent nature of BISNs introduces a new layer of complexity to maintaining fair packet delivery. Low energy availability impacts not only the throughput of the low-energy node, but can prevent other nodes from fairly communicating with the rest of the network. Future work could evaluate whether the definition for fair delivery should be modified to accommodate for intermittency and examine new energy-aware protocols to promote fairness in BISNs. Lure is well-suited to this work due to its intermittent foundation and intermittency-aware netstack.

5.3. A Shared Sense of Time (SST)

The time module of Lure is responsible for maintaining time across power failures and nodes in the network—i.e., developing a *shared sense of time*. Such time information is crucial in BISNs to ensure timely data collection and analysis [60], to increase communication probability [25], and to enable efficient distributed computation [61]. Timing information can be used for coordinated BISN behavior such as Clk-LMP [25], where two nodes turn off early at calculated times in an attempt to meet a specific future rendezvous for communication. Here we demonstrate Lure's capabilities when developing and evaluating a shared sense of time with our proposed *Levee* [62], an intermittency-aware version of Flooding Time Synchronization Protocol (FTSP) [63].

5.3.1. Research Question

Past work has identified that intermittency has a strong impact on communication between nodes [25]. A shared sense of time depends upon communication between nodes to propagate the (potentially limited) timing information available to each node. Thus, a critical question for BISN designers is: how does communication protocol choice impact the quality of time synchronization?

To investigate this question, we consider a 3 by 3 grid of batteryless wireless sensor nodes powered by an RF energy source at the top-left corner, as shown in Figure 18. We model the available RF energy as decreasing relative to the distance R from the source according to $1/R^2$ [64]. In this example, the resulting LCRs vary from about 0.25 to 0.01.

In FTSP, a root or reference node periodically floods the network with time information until all nodes in the network have synchronized their clocks. Root nodes elect themselves if they have the lowest node ID among all the nodes they have communicated to within a pre-determined time duration (i.e., node 0 will always believe it is the root node, while other nodes may sometimes believe they are the root due to severely intermittent communication). Generally, a time packet originates at the root node and floods outward to the other nodes in the network. Non-root nodes adjust their sense of time based on the time information contained in the incoming packets.

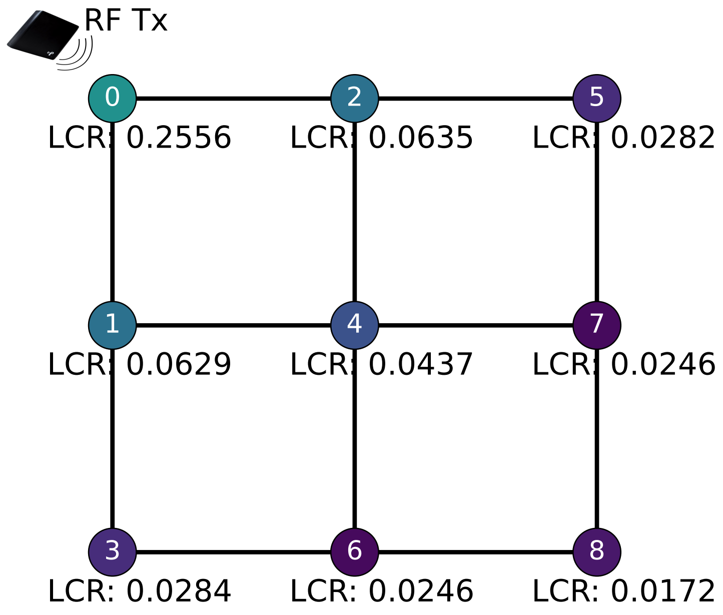


Figure 18: Example LCRs of sensor nodes in a 3 by 3 grid with an RF power source located at the top-left corner, near node 0.

5.3.2. Implementation

We implement a shared time class to model time synchronization in a BISN in Lure. This class provides a framework for adding new implementations of time synchronization protocols. To promote modularity and encapsulation, all shared time classes are sub-classes of `ContinuousTimeModule`—the default continuous time class. As such, a shared time protocol in Lure needs to override the `time` method, which provides the time since the beginning of simulation. In a shared time protocol, this method should provide a global shared sense of time across the network. Nodes share their time information with other nodes using the `Framer` interface. Finally, we add statistics to evaluate and debug a shared sense of time to `Stats`. Table 2 shows the additional effort in lines of code (LOC) for the additional classes needed to model the shared sense of time compared with the default time classes included in Lure. Overall, relatively few lines of additional code are required to add new shared sense of time approaches—all within a single new file per shared time approach.

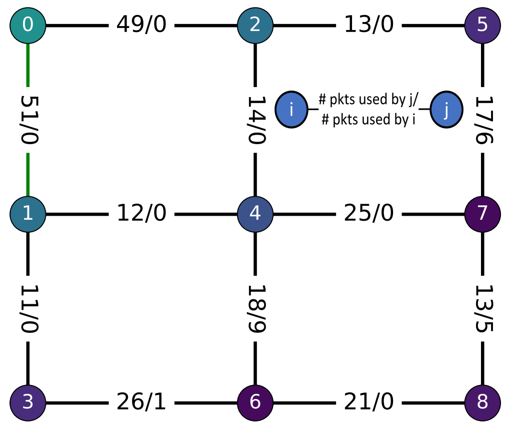
Class		**LOC**
`TimeModule`		91
`Clock`	`ActiveClock`	38
	`PersistentClock`	525
`ContinuousTimeModule`		91
Shared Sense of Time	`SSTFTSPTimeModule`	240
	`SSTLeveeTimeModule`	407
Additional Stats		6

Table 2: Development effort for shared sense of time.

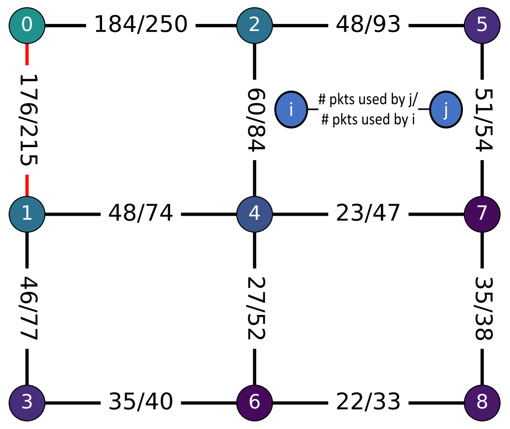
Lure also enables verification and debugging of implementations of shared sense of time. Recorded statistics such as estimated shared time, continuous clock time, and number of packets used to update a node's shared time can be visualized both in aggregate and per-node after the simulation ends. Figure 19 shows a per-node visualization that was used to debug an issue in an earlier version of our FTSP implementation.

The red connection in the upper left of Figure 19b shows that node 0 (root node) receives and uses 215 packets from node 1, while node 1 receives and uses 176 packets from node 0. This behavior in our earlier implementation is incorrect; nodes should filter out packets that were initiated by a higher ID root node (i.e., `msg_root_id > my_root_id`). Without this filtering, accuracy suffers since nodes may try to track multiple divergent root nodes.

Once this anomalous behavior is observed, the developer can go back into the FTSP module and make necessary changes to the condition where the filtering of incoming packets is done. After fixing the issue, we can see in Figure 19a that the successful communication from higher ID nodes to lower ID nodes has been reduced significantly and, importantly, node 0 never uses another node's timing packet. This specific instance is just one example of how Lure enables rapid verification and debugging of custom protocols for BISNs.



(a) Expected FTSP.



(b) Buggy FTSP.

Figure 19: Lure visualization of expected FTSP communication behavior compared to anomalous behavior.

5.3.3. Evaluation

To evaluate time synchronization in a BISN in Lure, we create a network of 9 batteryless nodes in a 3 by 3 grid. The energy harvesting source is placed at the top-left corner of the grid, similar to Figure 18. We vary the average lifecycle ratio of the network by changing the power coming out of the harvesting source. Nodes in the network communicate according to an adapted FTSP protocol where the root node sends time packets synchronously and other nodes asynchronously. We configure the nodes to use `BasicILL` and `FixedLMP`, with a communication slot length of 5 ms. Figure 20 shows the shared sense of time error ($M_{\text{lifecycle}}$ from [62], which allows any node that turns on within a measurement period to participate in the error metric) versus the average LCR of the network for different `FixedLMP` configurations. We report average steady-state values from 25 trials. From these results, we observe that all time measurement accuracies degrade with a lower average LCR, partially due to the longer off-times resulting in higher error from the persistent clock readings (i.e., worse local time error). The shared sense of time is then even more sensitive to LCR due to its dependence on degrading communication. This degradation can be mitigated using the LMP—we find that an FTSP-based shared sense of time has a 5× (at LCR of 0.02) and 4× (at LCR of 0.20) improvement using a `FixedLMP` configuration of two slots relative to the baseline where no LMP control is used (i.e., 50). Between `FixedLMP` configurations, as overlaps become more frequent and thus communication latency decreases, the error in the shared sense of time decreases. Still, the relatively large error at lower LCRs suggests that there is ample room for innovation to further improve the shared sense of time in BISNs.

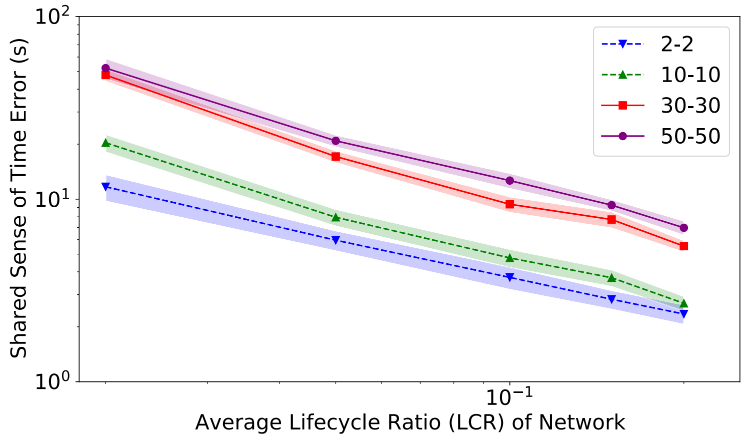


Figure 20: Shared sense of time error in a sensor network consisting of 9 batteryless nodes arranged in a 3 by 3 grid topology.

6. Conclusion

In conclusion, we have presented Lure, a new simulator for batteryless intermittent sensor networks. We have calibrated Lure against real-world batteryless intermittent systems, and shown that it supports a wide variety of research purposes, including explorations of intermittent behavior, novel LMP protocols, intermittent timekeeping solutions, specifications for novel intermittency-management hardware, and intermittency-aware broadcast and network protocols. For future work on Lure, we have identified a number of new features and modules that would expand Lure's capabilities as a research tool. Some of these planned features include more in-depth modeling of the intermittent application layer, more options for the physical layer in the network stack (such as backscatter radios), and support for trace-based energy model input. Alongside this paper, we are open-sourcing Lure, with the hope that it can aid the community in novel and exciting batteryless intermittent sensor network research.

Appendix A. Lure Description and Technical Details

In this section, we describe Lure's design goals, core components, environment energy modeling, intermittent node framework, and data collection and processing features.

Appendix A.1. Design Goals and Implementation Choices

Lure is foremost designed as a research tool for the batteryless intermittent communications community. It is intended to allow fast prototyping and evaluation of protocols and software modules at various layers in an intermittent communication and application stack. Lure is also designed to enable evaluation of novel hardware to support communication and computation on batteryless intermittent nodes. To achieve these goals, we adhered to the following high-level design philosophy and implementation choices.

Appendix A.1.1. Python

Lure is implemented in Python. Python is advantageous as a language choice because it is well-known among researchers and reasonably easy to learn. It is powerful, with many built-in language features useful for simulation, but also an extensive set of widely-used libraries for mathematical operations and plotting. We make full use of Python's type hinting library (`typing`), providing some measure of type safety and code autocompletion, while still allowing a researcher familiar with Lure to use Python's dynamic typing to bend the rules enough to customize Lure to their needs.

The use of Python has two potential limitations. First, Python is slow compared to fully-compiled languages such as C++. We mitigate this issue by using the popular SimPy [65] library to efficiently simulate only useful timesteps, and by baking in parallel processing (via Python's `multiprocessing` module).

Second, code written in Python cannot generally be compiled to run on the physical devices being tested. This means that protocols implemented in Lure must be reimplemented for physical deployments, and that unlike some tools such as Cooja, Lure is not a debugging tool for code intended for devices (though a feature like this could be added in the future). Since node architectures for BISNs are yet to be finalized, we accept this limitation as a tradeoff for making Lure more conducive to rapid, flexible prototyping.

Appendix A.1.2. Modularity

Lure is intended to be fully modular through object-oriented programming paradigms. A researcher new to Lure can implement and validate a design simply by subclassing one or more existing classes. These classes represent distinct components, described in later sections, with well-defined interfaces between them. These interfaces enable separation of concerns between components, and ensure that interactions between components are limited to that which is feasible in a real implementation. To verify the feasibility of our interfaces, we have simultaneously been developing a testbed of prototype physical nodes, powered solely by energy harvesting and running a software framework that mirrors that of Lure's `SensorNode`. This physical testbed contributes to the validation of Lure, which will be discussed in Section 4.

Appendix A.1.3. Configurability

Lure is completely configurable, meaning that any parameter of any component can be easily configured by the end user, with no hard-coded values. This is accomplished via an extensive JSON-based configuration system. In JSON (JavaScript Object Notation, chosen because it is human-readable and similar to built-in Python data structures), end users can define simulation experiments to be run, specifying the class of each component and its corresponding parameters.These JSON files are loaded by Lure when it runs Any components or values not specified will revert to defaults (also defined in JSON), reducing the need for extensive manual configuration.

Appendix A.2. Lure Core Components

Lure is designed with the experimenter in mind. Out of the box, it contains tools to run repeatable experiments and produce plots for a set of dependent variables versus any independent variable that can be configured using the JSON configuration system. The core components (Python objects) of Lure naturally map to this type of results . A Lure `Experiment` consists of a set of `DataSeries` objects, analogous to lines on a 2D line plot. Each design point in a `DataSeries` is a set of `Simulation` objects with different random seeds.

All of these components are encapsulated in a `Lure` object that runs all defined `Simulation` objects in parallel processes (with the number of processes configured in JSON). The results from each `Simulation` are written to separate subdirectories in an output directory, minimizing the memory consumption and allowing a `Lure` run to be interrupted and resumed at a later time. The `Lure` class provides a convenience method to load the results from the output directory and give them to the user in a concise, parseable form.

At the core of the `Simulation` object is SimPy, a popular event-driven simulation framework library for Python. All `SensorNode` objects, as well as the `EnergyModel` (described below), are SimPy "processes." In general, the `SensorNode` objects process many events during their on-times, and very few events during their off-times, resulting in minimal simulation overhead for intermittency (though the effect of intermittency on system performance still means that, for more severe intermittency, more simulated time may need to pass before e.g. a certain number of packets are sent).

Appendix A.3. Energy Model

Each `Simulation` has an `EnergyModel` object responsible for providing the instantaneous power available from the environment to a `SensorNode`. Depending on the `EnergyModel` implementation, this value may change over time and may be different for different nodes. Lure's default `EnergyModel` samples from a Gaussian distribution separately for each node. These samples are taken at regular update intervals, such that the energy harvesting rate of a node fluctuates over time. More advanced `EnergyModel` subclasses could take factors such as a node's position (as shown in Section 5.3) or the time of day into account.

Appendix A.4. Intermittent Node Components

A `Simulation` consists of a set of `SensorNode` objects. As the main component of the system under test, the `SensorNode` is a complex object. It is built around an intermittency-aware application framework and network stack[4]. As shown in Fig. 2, the current main components of a `SensorNode` are (1) a power supply, (2) a lifecycle management protocol (LMP) module, (3) an intermittent time module, (4) a application (often atraffic generator for simulations), and (5) an intermittency-aware network stack (netstack). To achieve the modularity goal discussed above, each of these components has template classes that define their external interfaces. A `SensorNode` object instantiates concrete subclasses of these templates according to the JSON configuration provided by the end user. Most components have the following common interface:

- An `initialize` method, with the `SensorNode` as a parameter. This is called once, before the simulation begins but after all components are instantiated (analogous to the deployment stage of a physical system). By default, components do not have access to other components in Lure. This is a design decision intended to encourage components to interact in a realistic and implementable manner. But since the `SensorNode` is passed in to `initialize`, this function allows a component to obtain a handle to any other component in the node. In this way we support forming arbitrary relationships between components, while nudging the component author to carefully consider these relationships.
- A `boot` method. This method is called every time the `SensorNode` boots (some amount of time after it turns on, configurable in order to simulate OS boot time overhead). The `boot` method allows the components to perform actions that need to be performed every time the node restarts, which is critical in simulating intermittency.
- An `execute` method. This method is called for every simulated event, allowing components to perform time-based actions.
- A pair of `get_config` and `set_config` methods. These methods use defined keys to retrieve or configure parameters of the component. This allows components to be dynamically reconfigured by other components, supporting complex interactions through a well-defined, but minimal, interface.

The five main components of `SensorNode` are discussed in more detail below.

Appendix A.4.1. Power Supply

We simulate the power management hardware of intermittent nodes with a `PowerSupply` object. A `PowerSupply` consists of a `Harvester` and at least one `Storage` object. Whenever simulation time advances, the `Harvester` harvests an amount of energy based on the current state of the `EnergyModel`. The `PowerSupply` stores this energy in the `Storage` (a `Capacitor` by default). If the `SensorNode` is on, it withdraws energy from the `PowerSupply` based on its consumption rate, which can vary with the state of the node. The `PowerSupply` is also responsible for enforcing the charge state of the `Storage` and determining when the `SensorNode` dies or restarts, based on the hardware being simulated. The default `PowerSupply` simulates a threshold-based voltage supervisor similar to [31], but we have also used this framework to simulate novel custom hardware such as [66]. We also note that this framework can easily be used to simulate a continuously-powered node.

[4]A paper further detailing this intermittency-aware network stack is currently under submission.

Appendix A.4.2. Life Cycle Management Protocol (LMP)

We implement the LMP as an `LMP` object belonging to the `SensorNode`. The `LMP` has an `enable` function (for turning on the LMP's control over lifecycling) and a `disable` function (for suspending control). Lure's default `LMP`, called `FixedLMP`, simulates hardware that allows the node to turn itself off at an arbitrary, but fixed, point within its on-time[5]. The `FixedLMP` protocol activates this capability after the node has been on for a certain (configurable) amount of time. We have also implemented an LMP that chooses a random on-time from a configured range, and an LMP designed to work in tandem with the custom `PowerSupply` hardware previously mentioned. Finally, we have implemented a soft intermittency framework [8] as an `LMP` subclass. This framework resembles that used in intermittent communication works such as [29]. While neither Lure nor the intermittency-aware framework it implements were initially designed with soft intermittency in mind, the ability to implement soft intermittency as an `LMP` in Lure demonstrates the power and wide applicability of both.

Appendix A.4.3. Time Module

In Lure, each clock is implemented as a separate object, and a `TimeModule`object is responsible for merging them into a single interface used by other components. This interface consists of a `clock` function that provides the (best estimate of the) time since the node last booted, and a `time` function that provides the (best estimate of the) continuous time. This may be a local continuous time, or a global "shared" sense of continuous time [62], depending on the `TimeModule` used (see Section 5.3).

This level of granularity for timekeeping is important so that Lure can evaluate the effect of intermittent-specific (as well as traditional) timekeeping phenomena on protocol performance. For example, an experiment could evaluate performance of a continuous-time-based communications protocol under varying levels of accuracy of the persistent clock. The `TimeModule` abstraction allows Lure to serve as a prototyping platform for intermittent timekeeping systems (e.g. new persistent clocks), and also for protocols that synchronize time across nodes in the network, a classic area of WSN/IoT research that may need to be revisited for intermittent systems [54, 62].

Appendix A.4.4. Traffic Generator

From an application-layer perspective, `SensorNode`s generate traffic via a `TrafficGenerator` object. This traffic is then sent by the netstack. Lure comes with a `TrafficGenerator` subclass that implements the common Poisson point process traffic generation pattern, and more advanced traffic patterns, as well as trace-based input, could easily be implemented using the `TrafficGenerator` interface. We leave development of a more detailed application layer, which could include sensing and computation modeling, to future work.

Appendix A.4.5. Intermittency-aware Network Stack

The remaining components described here compose an intermittency-aware network stack. In Lure, these components are objects that belong to a `Netstack`object. The `Netstack` also stores a registry of objects that implement a `Framer` interface. `Framer` objects are given the chance to add headers to, or parse headers out of, all packets sent or received by the `Netstack`. This allows components such as the `LMP` or the `TimeModule` to piggyback control or timing information on data traffic.

- *Network Layer:* Currently, Lure's default `Network` object supports single-hop networks and statically-defined multihop routes. Forming dynamic networks over intermittent links is an interesting area of future research; to prototype such a protocol, a researcher using Lure only needs to override a `get_next_hop` method.
- *Intermittent Link Layer (ILL):* Lure's default `ILL` class maintains a queue of packets that persists across off-times and resumes communication upon boot after an off-time. A more advanced `ILL` could predict when particular neighbors are going to be on and manipulate the `LMP` to try to increase the probability of overlap.

[5]Doing so saves energy, shortening the following off-time and increasing the frequency of the lifecycle, which can, for example, aid in communication performance [24].

The ILL is also responsible for advocating for communication to other intermittency-aware components of the node. For example, with no knowledge of events happening within the netstack, the `LMP` could turn the node off in the middle of a successful communication, even if the node has energy to remain on. To prevent this from happening, we implement an `ILL` subclass called `BasicILL` that disables the `LMP` (so that the `LMP` cannot turn the node off) when communication occurs, and re-enables the `LMP` when communication has finished.

- *Media Access Control (MAC):* Lure's `MAC` object handles packet-based communication between neighboring nodes. Lure's baseline intermittency-aware MAC protocol is similar to a traditional duty-cycled sender-initiated MAC protocol for WSNs (e.g. [67, 68]). When the channel is clear, a sending node repeatedly transmits its data packet until it is acknowledged by the receiver (or the node shuts down due to either the `LMP` or the `PowerSupply`). If the channel is not clear before transmitting, the node backs off and listens for an incoming packet. Whenever a packet is received, the node continues to listen for additional packets.

- *Physical Layer:* Lure's physical layer handles the transmission of `Packet` objects between nodes and establishes the physical topology of the simulation. On initialization, the `Physical` object of each node can be configured with a list of physical neighbors, or with cartesian coordinates and a communication range, among other options. All transmissions go through the `Physical` object, ensuring that nodes are only able to talk within their configured neighborhood. The `Physical` object also provides clear-channel assessment (CCA) functionality. We leave the development of a more detailed physical layer (e.g. path loss-based) to future work.

Appendix A.5. Data Collection and Processing

Lure provides a variety of flexible data collection and processing mechanisms.

Appendix A.5.1. Logging

Lure includes a custom logging module that creates a `Simulation`-wide log shared between all components implementing a `Loggable` interface. Users subclassing any of the existing `SensorNode` components have access to logging out of the box. Logged messages are saved in the output directory for the `Simulation`. Log messages automatically include the `SensorNode`'s ID, the component name, and a timestamp of simulated time. Components can log messages at the typical levels of importance—debug, info, warning, error, and critical—and the logging level can be configured on a component-by-component basis. We find the logs to be primarily useful for development, though their uniform message format is conducive to post-processing as well.

Appendix A.5.2. Stats

Lure's primary means of experimental data collection is the `Stats` module. Each `SensorNode` has a `Stats` object, which is essentially a key-value store. Components that implement the `StatsProvider` interface are able to track arbitrary values in their node's `Stats` object. The `Stats` module provides explicit support for stats stored as scalars, lists, or time series. After the `Simulation` finishes, Lure saves the `Stats` objects from all nodes to disk in the output directory using Python's pickle module. Lure provides a standard set of stats, such as a time series of communication events, as well as a `StatsParser` class for post-processing of these stats, such as calculating the communication delays. Additionally, component subclasses can track their own custom stats simply by using unique keys. Lastly, components may inherit the `StatsObserver` class to be notified of changes to stats in real time, allowing for additional data collection features.

Appendix A.5.3. Plotting

Lure comes with a `Plotter` module that can produce various plots of Lure results using the matplotlib package. `Plotter` is capable of plotting a set of standard metrics (including throughput, delay, and packet count) versus a set of typical independent variables (including lifecycle ratio and traffic rate). `Plotter` is also capable of plotting any set of time series from a `Stats` object, and the distribution of scalar stats over a set of simulations.

Acknowledgment

We would like to thank Shen Fu, Tyler Orman, and Tasman Grinnell for additional contributions. This work was supported in part by the U.S. National Science Foundation under Grants 1730275, 2008548, and 2144757.

References

[1] M. Hasan, State of IoT 2022: Number of connected IoT devices growing 18% to 14.4 billion globally, https://iot-analytics.com/number-connected-iot-devices//%7D (2022).

[2] ONiO, Better battery recycling is still no match for batteryless iot, https://www.onio.com/article/batteryless-vs-better-recycling-iot-future.html, accessed: 2024-04-2 (2 2024).

[3] W. Mrozik, M. A. Rajaeifar, O. Heidrich, P. Christensen, Environmental impacts, pollution sources and pathways of spent lithium-ion batteries, Energy & Environmental Science 14 (12) (2021) 6099–6121.

[4] A. J. Williams, M. F. Torquato, I. M. Cameron, A. A. Fahmy, J. Sienz, Survey of energy harvesting technologies for wireless sensor networks, IEEE Access 9 (2021) 77493–77510.

[5] J. Hester, J. Sorber, The future of sensing is batteryless, intermittent, and awesome, in: Proceedings of the 15th ACM Conference on Embedded Network Sensor Systems, 2017, pp. 1–6.

[6] S. Fu, V. Narayanan, M. L. Wymore, V. Deep, H. Duwe, D. Qiao, No battery, no problem: Challenges and opportunities in batteryless intermittent networks, Journal of Communications and Networks 25 (6) (2023) 806–813.

[7] S. Ahmed, B. Islam, K. S. Yildirim, M. Zimmerling, P. Pawełczak, M. H. Alizai, B. Lucia, L. Mottola, J. Sorber, J. Hester, The internet of batteryless things, Communications of the ACM 67 (3) (2024) 64–73.

[8] M. L. Wymore, H. Duwe, A tale of two intermittencies, in: Proceedings of the 11th International Workshop on Energy Harvesting & Energy-Neutral Sensing Systems, 2022, pp. 1–3.

[9] H. Jayakumar, A. Raha, V. Raghunathan, Quickrecall: A low overhead hw/sw approach for enabling computations across power cycles in transiently powered computers, in: 2014 27th International Conference on VLSI Design and 2014 13th International Conference on Embedded Systems, IEEE, 2014, pp. 330–335.

[10] D. Balsamo, A. S. Weddell, G. V. Merrett, B. M. Al-Hashimi, D. Brunelli, L. Benini, Hibernus: Sustaining computation during intermittent supply for energy-harvesting systems, IEEE Embedded Systems Letters 7 (1) (2014) 15–18.

[11] K. Maeng, B. Lucia, Adaptive dynamic checkpointing for safe efficient intermittent computing, in: 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18), 2018, pp. 129–144.

[12] D. Balsamo, A. S. Weddell, A. Das, A. R. Arreola, D. Brunelli, B. M. Al-Hashimi, G. V. Merrett, L. Benini, Hibernus++: a self-calibrating and adaptive system for transiently-powered embedded devices, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 35 (12) (2016) 1968–1980.

[13] A. Y. Majid, C. D. Donne, K. Maeng, A. Colin, K. S. Yildirim, B. Lucia, P. Pawełczak, Dynamic task-based intermittent execution for energy-harvesting devices, ACM Transactions on Sensor Networks (TOSN) 16 (1) (2020) 1–24.

[14] M. Karimi, H. Choi, Y. Wang, Y. Xiang, H. Kim, Real-time task scheduling on intermittently powered batteryless devices, IEEE Internet of Things Journal 8 (17) (2021) 13328–13342.

[15] B. Ransford, J. Sorber, K. Fu, Mementos: System support for long-running computation on rfid-scale devices, in: Proceedings of the sixteenth international conference on Architectural support for programming languages and operating systems, 2011, pp. 159–170.

[16] K. Maeng, A. Colin, B. Lucia, Alpaca: Intermittent execution without checkpoints, arXiv preprint arXiv:1909.06951 (2019).

[17] V. Deep, V. Narayanan, M. Wymore, D. Qiao, H. Duwe, HARC: A heterogeneous array of redundant persistent clocks for batteryless, intermittently-powered systems, in: 2020 IEEE Real-Time Systems Symposium (RTSS), IEEE, 2020, pp. 270–282.

[18] J. Hester, N. Tobias, A. Rahmati, L. Sitanayah, D. Holcomb, K. Fu, W. P. Burleson, J. Sorber, Persistent clocks for batteryless sensing devices, ACM Transactions on Embedded Computing Systems (TECS) 15 (4) (2016) 1–28.

[19] J. de Winkel, C. Delle Donne, K. S. Yildirim, P. Pawełczak, J. Hester, Reliable timekeeping for intermittent computing, in: Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems, 2020, pp. 53–67.

[20] V. Narayanan, R. Sahu, J. Sun, H. Duwe, BOBBER a prototyping platform for batteryless intermittent accelerators, in: Proceedings of the 2023 ACM/SIGDA International Symposium on Field Programmable Gate Arrays, 2023, pp. 221–228.

[21] G. Gobieski, B. Lucia, 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, 2019, pp. 199–213.

[22] H. Desai, M. Nardello, D. Brunelli, B. Lucia, Camaroptera: A long-range image sensor with local inference for remote sensing applications, ACM Transactions on Embedded Computing Systems (TECS) 21 (3) (2022) 1–25.

[23] Q. Huang, Y. Mei, W. Wang, Q. Zhang, Toward battery-free wearable devices: The synergy between two feet, ACM Transactions on Cyber-Physical Systems 2 (3) (2018) 1–18.

[24] V. Deep, M. L. Wymore, A. A. Aurandt, V. Narayanan, S. Fu, H. Duwe, D. Qiao, Experimental study of lifecycle management protocols for batteryless intermittent communication, in: 2021 IEEE 18th International Conference on Mobile Ad Hoc

[25] M. L. Wymore, V. Deep, V. Narayanan, H. Duwe, D. Qiao, Lifecycle management protocols for batteryless, intermittent sensor nodes, in: 2020 IEEE 39th International Performance Computing and Communications Conference (IPCCC), IEEE, 2020, pp. 1–8.

[26] Madisound, Mdl 50 mfd non-polar electrolytic capacitor 100v, https://www.madisoundspeakerstore.com/m.d.l.-electrolytic-capacitor-100vdc/mdl-50-mfd-non-polar-electrolytic-capacitor-100v/ (2024).

[27] A. Y. Majid, M. Jansen, G. O. Delgado, K. S. Yildirim, P. Pawełłzak, Multi-hop backscatter tag-to-tag networks, in: IEEE INFOCOM 2019-IEEE Conference on Computer Communications, IEEE, 2019, pp. 721–729.

[28] A. Torrisi, K. S. Yıldırım, D. Brunelli, Reliable transiently-powered communication, IEEE Sensors Journal 22 (9) (2022) 9124–9134. doi:10.1109/JSEN.2022.3158736.

[29] K. Geissdoerfer, M. Zimmerling, Learning to communicate effectively between battery-free devices, in: 19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22), USENIX Association, Renton, WA, 2022, pp. 419–435.

[30] T. Zhu, J. Li, H. Gao, Y. Li, Broadcast scheduling in battery-free wireless sensor networks, ACM Transactions on Sensor Networks (TOSN) 15 (4) (2019) 1–34.

[31] Powercast, P2110B 915 MHz RF Powerharvester Receiver (12 2016).

[32] E. Dallago, A. L. Barnabei, A. Liberale, G. Torelli, G. Venchi, A 300-mv low-power management system for energy harvesting applications, IEEE Transactions on Power Electronics 31 (3) (2015) 2273–2281.

[33] M. Gshash, V. Narayanan, H. Duwe, N. M. Neihart, Rf energy harvester with constant off-time charger for batteryless devices, in: 2023 IEEE 66th International Midwest Symposium on Circuits and Systems (MWSCAS), IEEE, 2023, pp. 259–263.

[34] Texas Instruments, MSP430FR58xx, MSP430FR59xx, and MSP430FR6xx Family User's guide (10 2017).

[35] Texas Instruments, LAUNCHXL-CC1352R1 CC1352R LaunchPad™ development kit for SimpleLink™ multi-band wireless MCU (09 2019).

[36] F. Osterlind, A. Dunkels, J. Eriksson, N. Finne, T. Voigt, Cross-level sensor network simulation with Cooja, in: Proceedings. 2006 31st IEEE conference on local computer networks, IEEE, 2006, pp. 641–648.

[37] P. Levis, S. Madden, J. Polastre, R. Szewczyk, K. Whitehouse, A. Woo, D. Gay, J. Hill, M. Welsh, E. Brewer, et al., TinyOS: An operating system for sensor networks, in: Ambient intelligence, Springer, 2005, pp. 115–148.

[38] G. F. Riley, T. R. Henderson, The ns-3 network simulator, in: Modeling and tools for network simulation, Springer, 2010, pp. 15–34.

[39] M. Capuzzo, C. Delgado, J. Famaey, A. Zanella, An ns-3 implementation of a battery-less node for energy-harvesting internet of things (2021).

[40] A. Varga, Omnet++, in: Modeling and tools for network simulation, Springer, 2010, pp. 35–59.

[41] E. Longman, M. El-Hajjar, G. V. Merrett, Intermittent opportunistic routing components for the inet framework, in: Proceedings of the 8th OMNeT++ Community Summit, 2021.

[42] E. Longman, M. El-Hajjar, G. V. Merrett, Multihop networking for intermittent devices, in: Proceedings of the 10th International Workshop on Energy Harvesting & Energy-Neutral Sensing Systems (ENSsys), 2022, pp. 878–884.

[43] A. Torrisi, F. Baggio, D. Brunelli, Transiently-powered batteryless device-to-device communication protocol simulator, in: R. Berta, A. De Gloria (Eds.), Applications in Electronics Pervading Industry, Environment and Society, Springer Nature Switzerland, Cham, 2023, pp. 279–286.

[44] B. Lucia, B. Ransford, A simpler, safer programming and execution model for intermittent systems, ACM SIGPLAN Notices 50 (6) (2015) 575–585.

[45] J. San Miguel, K. Ganesan, M. Badr, C. Xia, R. Li, H. Hsiao, N. E. Jerger, The eh model: Early design space exploration of intermittent processor architectures, in: 2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO), IEEE, 2018, pp. 600–612.

[46] M. Furlong, J. Hester, K. Storer, J. Sorber, Realistic simulation for tiny batteryless sensors, in: Proceedings of the 4th International Workshop on Energy Harvesting and Energy-Neutral Sensing Systems, 2016, pp. 23–26.

[47] S. C. Wong, S. T. Sliper, W. Wang, A. S. Weddell, S. Gauthier, G. V. Merrett, Energy-aware hw/sw co-modeling of batteryless wireless sensor nodes, in: Proceedings of the 8th International Workshop on Energy Harvesting and Energy-Neutral Sensing Systems, 2020, pp. 57–63.

[48] J. Göpfert, B.-C. Renner, High-level simulation of the timely behavior of intermittent systems, in: Proceedings of the 11th International Workshop on Energy Harvesting & Energy-Neutral Sensing Systems, 2023, pp. 1–7.

[49] H. Rocha, G. Korol, M. Jordan, A. Krause, R. Silveira, C. Vieira, P. Navaux, G. L. Nazar, L. Carro, A. C. S. Beck, Firefly: An open-source rocket-based intermittent framework, in: 2020 33rd Symposium on Integrated Circuits and Systems Design (SBCCI), IEEE, 2020, pp. 1–6.

[50] E. Gelenbe, Energy packet networks: adaptive energy management for the cloud, in: Proceedings of the 2nd International Workshop on Cloud Computing Platforms, 2012, pp. 1–5.

[51] Y. A. El Mahjoub, J.-M. Fourneau, H. Castel-Taleb, Energy packet networks with general service time distribution, in: 2020 28th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS), IEEE, 2020, pp. 1–8.

[52] E. Gelenbe, A sensor node with energy harvesting, ACM SIGMETRICS Performance Evaluation Review 42 (2) (2014) 37–39.

[53] E. Gelenbe, Y. Zhang, Performance optimization with energy packets, IEEE Systems Journal 13 (4) (2019) 3770–3780.

[54] E. Çürük, K. S. Yıldırım, P. Pawelczak, J. Hester, On the accuracy of network synchronization using persistent hourglass clocks, in: Proceedings of the 7th International Workshop on Energy Harvesting & Energy-Neutral Sensing Systems, 2019, pp. 35–41.

[55] E. Gelenbe, E. T. Ceran, Energy packet networks with energy harvesting, IEEE access 4 (2016) 1321–1331.

[56] J. Hester, T. Scott, J. Sorber, Ekho: Realistic and repeatable experimentation for tiny energy-harvesting sensors, in: Proceedings of the 12th ACM Conference on Embedded Network Sensor Systems, 2014, pp. 330–331.

[57] Texas Instruments, MSP430FR5994 LaunchPad™ Development Kit (MSP-EXP430FR5994) (09 2019).

[58] Powercast, P2110B 915 MHz RF Powerharvester Receiver (12 2016).

[59] C. T. Ee, R. Bajcsy, Congestion control and fairness for many-to-one routing in sensor networks, in: Proceedings of the 2nd international conference on Embedded networked sensor systems, 2004, pp. 148–161. doi:10.1145/1031495.1031513.

[60] J. Hester, K. Storer, J. Sorber, Timely execution on intermittently powered batteryless sensors, in: Proceedings of the 15th ACM Conference on Embedded Network Sensor Systems, 2017, pp. 1–13.

[61] R. Sahu, R. Toepfer, M. D. Sinclair, H. Duwe, Denni: Distributed neural network inference on severely resource constrained edge devices, in: 2021 IEEE International Performance, Computing, and Communications Conference (IPCCC), IEEE, 2021, pp. 1–10.

[62] V. Deep, M. L. Wymore, D. Qiao, H. Duwe, Toward a shared sense of time for a network of batteryless, intermittently-powered nodes, in: 2022 IEEE International Performance, Computing, and Communications Conference (IPCCC), IEEE, 2022, pp. 138–146.

[63] M. Maróti, B. Kusy, G. Simon, A. Lédeczi, The flooding time synchronization protocol, in: Proceedings of the 2nd International Conference on Embedded Networked Sensor Systems, 2004, pp. 39–49.

[64] S. K. Divakaran, D. D. Krishna, Nasimuddin, RF energy harvesting systems: An overview and design issues, International Journal of RF and Microwave Computer-Aided Engineering 29 (1) (2019) e21633.

[65] SimPy: Discrete event simulation for Python, https://simpy.readthedocs.io/en/latest/.

[66] M. Gshash, V. Narayanan, H. Duwe, N. M. Neihart, RF energy harvester with constant off-time charger for batteryless devices, in: 2023 IEEE 66th International Midwest Symposium on Circuits and Systems (MWSCAS), 2023.

[67] M. Buettner, G. V. Yee, E. Anderson, R. Han, X-MAC: a short preamble MAC protocol for duty-cycled wireless sensor networks, in: Proc. ACM SenSys, 2006.

[68] A. Dunkels, The ContikiMAC radio duty cycling protocol, Tech. Rep. 5128, SICS (Jan. 2012).