Grace: Low-Cost Time-Synchronized GPIO Tracing for IoT Testbeds

Laura Harms[1,2], Christian Richter[1], Olaf Landsiedel[1,2]
[1]*Kiel University, Germany*
[2]*Chalmers University of Technology, Sweden*
lah@informatik.uni-kiel.de, stu204011@mail.uni-kiel.de, ol@informatik.uni-kiel.de

***Abstract*—Testbeds have become a vital tool for evaluating and benchmarking applications and algorithms in the Internet of Things (IoT). Testbeds commonly consist of low-power IoT devices augmented with observer nodes providing control, logging, and often also power-profiling. Today, the research community operates numerous testbeds, sometimes with hundreds of IoT nodes, to allow for detailed and large-scale evaluation. Most testbeds, however, lack opportunities for tracing distributed program execution with high accuracy in time, for example, via minimally invasive, distributed GPIO tracing. And the ones that do, like Flocklab, are built from custom hardware, which is often too complex, inflexible, or expensive to use for other research groups.**

This paper closes this gap and introduces *Grace*, a low-cost, retrofittable, distributed, and time-synchronized GPIO tracing system built from off-the-shelf components, costing less than €20 per node. *Grace* extends observer nodes in a testbed with (1) time-synchronization via wireless sub-GHz transceivers and (2) logic analyzers for GPIO tracing and logging, enabling time-synchronized GPIO tracing at a frequency of up to 8 MHz. We deploy *Grace* in a testbed and show that it achieves an average time synchronization error between nodes of 1.53 µs.

***Index Terms*—GPIO Logging, GPIO Tracing, Testbed, Internet of Things, IoT, Time-Synchronization**

I. Introduction

With more than 10 billion connected IoT devices deployed by the end of 2021 [1], the Internet of Things enables new applications in our connected and data-driven society. Their connected and often distributed nature makes extensive testing, evaluation, and benchmarking a must to ensure proper performance.

Simulation [2], [3] allows for high-level insights into protocols and algorithms. However, simulation cannot capture all details of the environment, nor is it capable of evaluating the performance of protocols on real hardware. Therefore, the research community commonly uses testbeds: deployments of (low-power) IoT devices co-located with observer infrastructure, typically an edge device – like a Raspberry Pi – for instrumentation, logging, and deployment control.

While testbeds provide real-world insights and are today's established tool for evaluating distributed IoT applications, they lack one essential capability: The capability to non-intrusively – or with minimal intrusiveness – track the execution of distributed protocols and algorithms. For example, for debugging and evaluating protocols, we often need insight into the execution and states within the hardware. For non-distributed settings such as traditional software development, one commonly uses debuggers, with which one can halt program execution and inspect the system's state. In distributed settings, we cannot halt the operation of nodes as both the environment continues to change, and further nodes will also continue their operation. Another common way is printing messages during operation, usually through a serial interface. However, printing takes several tens of milliseconds, which leads to side effects on program executions and limits accurate timestamping. It might even break the timing in timing-critical sections of a program, leading to missed deadlines.

The third way of gaining insight is through tracing the General-Purpose Input/Output (GPIO) pins of a processor or microcontroller. Toggling GPIO pins offers a minimally intrusive way of communicating timing-correct information on the operation to the outside world. A logic analyzer can record the GPIO traces to evaluate these later. While logic analyzers provide us with a time-accurate trace of the execution of a program, they commonly only provide insights into one device due to the physical distance between devices. However, in a distributed communication system, it is of high importance to know how multiple devices interact with each other and at what exact point in time, or how much time passes between the same operation on multiple devices. For example, Time-Division Multiple Access (TDMA) protocols like Glossy [4] or Time-Slotted Channel Hopping (TSCH) [5] are time-critical protocols that synchronize their communication; and among others, LWB [6] and Chaos [7] enable multiple devices to send data concurrently in a time-synchronized fashion. To evaluate synchronization of protocols like these and interaction between multiple devices, we require a GPIO tracing system, which performs a time-synchronized tracing on all devices. Many means of time-synchronization exist (including the Network Time Protocol (NTP) and the Global Positioning System (GPS)), however, none of them offers a low-cost, low complexity solution that is both available at indoor testbed locations and offers the required accuracy.

There are distributed, time-synchronized GPIO logging systems implemented in existing testbeds [8]–[10]. However, they use custom hardware with FPGAs and require a specific testbed observer platform throughout the testbed. This limits their adoption into other testbeds, especially those that already exist and use different hardware and observer platforms.

In this paper, we present *Grace*, a low-cost, retrofittable, distributed, and time-synchronized GPIO tracing system using

off-the-shelf components. *Grace* extends observer nodes with (1) time-synchronization via wireless sub-GHz (433 MHz) transceivers and (2) logic analyzers for GPIO tracing and logging. Using sub-GHz wireless, *Grace* enables a building-wide time-synchronization from a single central node performing unidirectional RBS-like time-synchronization. Further, we devise a software framework to enable extensive tracing capabilities building on this hardware. In our evaluation, we show that *Grace* is capable of continuously logging sparse data as produced when debugging IoT systems, such as wireless protocols, at a rate of 8 MHz. Moreover, we show that we achieve a time-synchronization of on average 1.53 µs, which, as we argue, is sufficient for most applications.

Overall, this paper makes the following contributions:

- We present *Grace*, a low-cost time-synchronized GPIO tracing system for IoT testbeds.
- We implement *Grace* using off-the-shelf hardware to enable easy adoption in other building-wide testbeds and make both the software and the hardware setup openly[1] available.
- We show *Grace*'s low cost of less than €20 per node.
- We evaluate *Grace*, showing its degree of time-synchronization between nodes of on average 1.53 µs, while not exceeding a worst-case synchronization of 3.75 µs.

The remainder of this paper is organized as follows. Section II gives necessary background information. Section III introduces *Grace*'s design, and Section IV presents our experimental evaluation. Section V reviews related GPIO tracing testbed systems, followed by the conclusion in Section VI.

II. Background

This section provides the necessary background for the remainder of this paper. We introduce (A) the concept of time synchronization with a focus on (B) the Network Time Protocol (NTP) and (C) the Reference Broadcasting System (RBS). Afterward, we provide general background on (D) Logic Analyzers.

A. Time Synchronization

Most electronic computing devices use a crystal oscillator as a basis for their clock. These oscillators operate at a certain frequency, but usually do not perfectly hold their nominal frequency. No oscillator is perfect, and physical variations like temperature or air pressure add to the oscillators' frequency variation. While these drifts are negligible in stand-alone single computer setups, they impose a challenge on distributed computing and communication systems. These systems require a tight synchronization of the individual clocks. For time-sensitive applications, these clocks have to fulfil one or both of these metrics: *precision* and *accuracy*. The notion of *precision* ( ) defines the maximum time error between two clocks (p, q) of a system ($\forall t; \forall p; q : jC_p(t) \quad C_q(t)j$). The notion of *accuracy* ( ) describes a clock's difference towards a reference timescale ($\forall t; \forall p : jC_p(t) \quad tj$) [11]. A common reference timescale is UTC, which the Network Time Protocol (NTP) (see section II-B) uses.

Depending on the importance of accuracy or precision, different synchronization approaches like NTP (see section II-B) or the Reference Broadcasting System (RBS) (see section II-C) are possible. Within the next sections, we describe these two in more detail.

B. Network Time Protocol (NTP)

The most widely used time synchronization protocol for distributed systems is the Network Time Protocol (NTP) [11]. It is the default protocol used by computers and most devices directly connected to the Internet, and builds the baseline for other protocols. For time synchronization, a device contacts an NTP server to receive the server's local time. From the received timestamps and the device's local timestamps of sending the request and receiving the response, the device can compute the round trip time and thus determine its offset from the reference clock.

In NTP, clocks are synchronized to UTC. As not each NTP server can be equipped with a reference clock, e.g., a GPS receiver, NTP builds a hierarchical structure of servers. This structure uses so-called stratum levels. A stratum-1 server is equipped with a reference clock. Each server is a stratum-$(k + 1)$ server if the server it contacts to synchronize to is a stratum-k server. NTP is known to achieve accuracy between 1 and 50 ms.

C. Reference Broadcasting System (RBS)

The reference broadcasting system (RBS) [11] differs significantly from methods like NTP. It does not assume the existence of an accurate clock (e.g., a UTC clock) within the network. Instead, it merely has the goal of network-internal clock synchronization. RBS is a wireless, physical layer time synchronization method. Moreover, contrary to other methods, where a node contacts a timeserver, in RBS, a time source broadcasts a reference signal to all nodes within the network. Every node generates a timestamp with its local clock on reception of the synchronization signal. As RBS only has a single sender that reaches all receivers, most parts of the critical path are eliminated. In a wireless network, the transmission time to all receivers is roughly the same, with a negligible offset. Thus, the critical part is only the reception (and timestamping) of the broadcast packet at the receivers. To reduce jitter introduced upon reception, RBS performs multiple broadcast rounds, and nodes exchange each other's delivery times to estimate their mutual, relative offset.

D. Logic Analyzer

A logic analyzer records the physical state of one or more signals over time. Logic Analyzers commonly trace digital signals. To process the trace of a logic analyzer, several software solutions working with logic analyzers have built-in features to not only display the recorded traces, but even decode protocols like the Serial Peripheral Interface (SPI)

[1] Available as open-source at: https://github.com/ds-kiel/grace

communication protocol to use a logic analyzer to debug communication between electronic components [12].

III. Design

Our design of *Grace* enables the time-synchronized use of GPIO tracing in building-wide testbeds while only using off-the-shelf components. Actuating GPIO pins is minimally intrusive and thus has negligible influence on a system's timing.

A. Design Overview

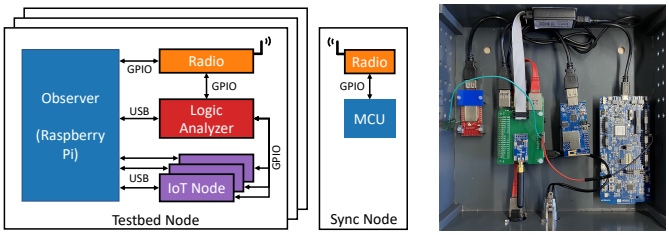
The diagram in Fig. 1a illustrates the general idea of *Grace*. One synchronization node repeatedly sends out a time signal. Each testbed node, equipped with a receiver, receives this time signal. Using this signal, each observer synchronizes its GPIO tracing clock. This synchronization of the tracing clocks enables the post-processing to match the GPIO traces of the different devices and build a common trace over all distributed devices.

Within the following sections, we describe the design of the different components in more detail. We discuss our time-error correction algorithm and discuss the system's integration into an existing testbed.

B. Synchronization Node

The synchronization node is a node (physically) independent of the testbed and its nodes. It consists of a microcontroller and a 433 MHz radio (see Fig. 1a). We use the 433 MHz band, which is an ISM band available in the International Telecommunication Union's (ITU) region 1 (i.a., Europa and Africa). Creating a design with 433 MHz radios allows the use of only one synchronization node for a typical building scale testbed due to its more extended range compared to the range of radios using higher frequencies. Moreover, the 433 MHz band is well outside the bands usually used for IoT research: the 2.4 GHz and 868/915 MHz bands. For larger testbeds beyond building scale, the use of multiple of these synchronization nodes is possible, but it requires them to be disciplined with an external clock (e.g., a GPS pulse per second signal) to ensure synchronization between the individual synchronization nodes.

According to a configurable time interval, in our case one second by default, the node generates and transmits a timestamp. This timestamp contains a counter value representing the time that has passed since turning on the synchronization node. As long as we only have one synchronization node, we do not require a globally accurate clock, but rather a single time source (here a microcontroller) within the vicinity of the network. The radio broadcasts this timestamp to all testbed nodes, which have to be in range of the synchronization node at all times. This approach of sending a timestamp at a regular interval (e.g., once a second) to all nodes of the network in a single(-hop) wireless broadcast follows the approach of the reference broadcasting system (RBS) (see Section II-C). Through the single broadcast and the close distance of all nodes to the synchronization node, the signal propagation delay is low enough that all nodes will receive the time signal with a negligible time offset of less than one logic analyzer sample or up to a few samples for physically large testbeds. Note that we do not intend to synchronize the clocks of testbed observers but rather the clocks corresponding to the logic analyzers.



(a) *Grace*'s node types.

(b) A testbed node.

Fig. 1: Design Overview of *Grace*.

C. Testbed node

A testbed node consists of a controller (e.g., a Raspberry Pi) and one or more low-power IoT devices as target platforms (see Fig. 1a). The target platforms expose GPIO pins that are to be traced. To allow this tracing, we build a system consisting of a USB logic analyzer and a radio that can be retrofitted to any testbed. We intend to use the logic analyzer for the GPIO tracing and the radio for receiving the timestamp from the synchronization node. We reserve one of the logic analyzer's pins for the radio. All other GPIO pins are available for tracing the target platforms' GPIO pins. Once the radio receives a signal from the synchronization node, it turns on its GPIO pin connected to the logic analyzer. This notifies the process, handling the logic analyzer's input of a new timestamp. Moreover, it pinpoints the reception of the time signal to an exact tick of the logic analyzer. In other words, we can match the reception time of the synchronization signal to a local timestamp of the GPIO tracing system. This allows us to perform error correction on the local time and thus have a notion of synchronization for combining the recorded traces of different devices in post-processing.

Within the following sections, we describe the different components of the GPIO tracing and the time synchronization.

D. GPIO Tracing

For GPIO tracing, *Grace* employs a USB-driven logic analyzer. This logic analyzer has to be able to trace sparse amounts of data on multiple GPIO pins and write the traces without prior processing to the USB buffer.

E. Trace Data Processing

The algorithms at the observer processes the incoming data in bulks. We perform different actions based on the changes present in each data sample. Each data sample is one recording of the logic analyzer. We compare each sample to its previous one. If there are changes present, we identify the corresponding channels of the logic analyzer. Depending on

Algorithm 1 Tick

```
Input: TICKS_PER_SECOND, state_clock, freq_adj, offset_adj,
    error_remaining, seconds, ticks_actual, accumulator
Output: offset_adj, error_remaining, seconds, ticks_actual, accumulator
 1: if state_clock = OFFSET or state_clock = FREQ then
 2:     accumulator ← accumulator + freq_adj − offset_adj
 3:     if offset_adj < 0 then
 4:         error_remaining ← error_remaining + offset_adj
 5:     else
 6:         error_remaining ← error_remaining − offset_adj
 7:     end if
 8:     if error_remaining ≠ 0 then
 9:         offset_adj = 0
10:     end if
11:     if accumulator > TICKS_PER_SECOND then
12:         seconds ← seconds + 1
13:         accumulator ← accumulator − TICKS_PER_SECOND
14:     end if
15:     ticks_actual ← ticks_actual + 1
16:     return
17: end if
```

Algorithm 2 Handle reference time signal

```
Input: WEIGHT, TICKS_PER_SECOND, state_clock, ticks_nominal, ticks_actual,
    seconds, freq_nominal, accumulator
Output: state_clock, freq_adj, offset, offset_adj, error_remaining, seconds,
    seconds_previous, ticks_actual
 1: seconds_ref ← read timestamp from radio
 2: if state_clock = WAIT then
 3:     seconds ← seconds_ref
 4:     state_clock ← OFFSET
 5: else
 6:     factor ← ((seconds_ref − seconds_previous) · ticks_nominal) / ticks_actual
 7:     if state_clock = OFFSET then
 8:         freq_adj ← freq_nominal · factor
 9:         state_clock ← FREQ
10:     else
11:         freq_adj ← (1 − WEIGHT) · freq_adj + WEIGHT · freq_nominal · factor
12:     end if
13:     if seconds_ref = seconds + 1 then
14:         error_remaining ← TICKS_PER_SECOND − accumulator
15:         offset ← accumulator − TICKS_PER_SECOND
16:     else if seconds_ref = seconds then
17:         error_remaining ← accumulator
18:         offset ← accumulator
19:     end if
20:     offset_adj ← offset = ticks_nominal
21:     ticks_actual ← 0
22: end if
23: seconds_previous ← seconds_ref
24: return
```

the channel's role, we perform further actions. If the state of the channel corresponding to the radio changed and that pin turned on, we know that we received a new time signal. We describe the algorithm for processing this time data in the following section (Section III-E1). If we detect a change on one of the traced GPIO pins, we timestamp the event (if we previously received at least two global timestamps) according to Equation 1 and hand it over for further processing. Lastly, we perform one clock tick of the logic analyzer updating its timestamp for the next sample.

$$\text{timestamp} = 10^9 \cdot \text{seconds} + \frac{\text{accumulator}}{\text{ticks per ns}} \quad (1)$$

For simplicity and without losing generality, we assume a time-stamp interval of one second for the algorithms. An adaptation to send out timestamp values representing a different timescale is also easily realizable.

1) Time Error Correction: Once the observer's tracing data processing algorithm detects the reception of a new time signal, we execute Algorithm 2. This algorithm essentially determines the time increment added for each sample recorded by the logic analyzer. At first, it reads the received data (timestamp) from the radio and sets the radio back into receive mode, which prepares the radio for receiving the next timestamp and turns off the radio's GPIO pin. Now we have the global timestamp and the exact tick it was received on. The processing of it differs depending on the state the GPIO tracing clock-correction system is in. The clock correction has three different states: WAIT, OFFSET, and FREQ. Initially, we start in state WAIT until we process our first time signal.

When receiving the first time signal, the algorithm saves the received timestamp as the current time with respect to the current sample, and as the previous timestamp for the algorithm's next iteration. Moreover, it changes the clock's state to OFFSET.

When the system is in state OFFSET or FREQ, we start by calculating a factor after receiving and reading the reference time ($seconds_{ref}$):

$$\text{factor} = \frac{(seconds_{ref} - seconds_{previous}) \cdot ticks_{nominal}}{ticks_{actual}} \quad (2)$$

This factor determines how much faster or slower the logic analyzer clock ran since receiving the previous timestamp. It uses both the current timestamp ($seconds_{ref}$) and the previously received timestamp ($seconds_{previous}$), as well as the nominal number of ticks ($ticks_{nominal}$) that should pass within a second (e.g. 8 000 000 at 8 MHz) and the actual number of ticks passed since the previous reception of a timestamp ($ticks_{actual}$). If this factor is 1, the clock of the logic analyzer is running at its nominal frequency. If the factor is < 1, the logic analyzer's sampling frequency is too high and if the factor is > 1, its frequency is too low. Using this factor, we can adjust the frequency value:

$$freq_{adj} = freq_{nominal} \cdot factor \quad (3)$$

If the clock's state is FREQ, we perform a slightly different frequency adjustment. We take a weighted approach between the previous frequency and the newly calculated frequency. That means, that we keep parts of the current frequency and only adjust it to a certain percentage. This way, we take previous changes into account and do not only react to the most recent time interval. This approach is comparable to a PI controller, taking the current offset and the history into account [13].

2) Phase offset: When the clock's state is OFFSET or FREQ, the algorithm performs an additional phase offset adjustment. If the logic analyzer's frequency is not exactly the nominal frequency and neither a multiple of it, there will remain an offset of ticks the logic analyzer's clock is ahead or behind the reference clock. This offset is a phase offset, which we also

need to handle when increasing our clock. To be able to correct this phase offset, we calculate the error to the closest second and an adjusted offset to adjust the phase when increasing the tick counter for each logic analyzer sample. Lastly, we reset the tick counter ($ticks_{actual}$) to zero.

If the current clock state is OFFSET, we change the clock's state to FREQ. Independent of the clock's state, we save the previous timestamp for the algorithm's next iteration.

2) Clock Tick: Algorithm 1 performs the frequency and phase adjustments computed in Algorithm 2 and discussed above. This algorithm only executes once the system has received an initial time value and thus is either in state OFFSET or FREQ. On each tick, we increase an accumulator value by a frequency value deduced by a phase offset value. With this, we increase the logic analyzer timestamp (cf. Equation 1) by a value close to its actual frequency while removing the phase error over time. This method of error correction corrects the error evenly spread over the course of one second. If we do not have a (remaining) phase error, the timestamp increases with the logic analyzer's frequency. Once the accumulator reaches the value corresponding to a second, we increase the second counter and reduce the accumulator by the respective value corresponding to one second. Lastly, we increase the tick counter.

This method of performing a tick at each sample has the advantage that it allows us to precisely adjust the clock of the logic analyzer. Moreover, the advantage of doing these adjustments in software is the granularity with which we can adjust phase and frequency. Instead of performing a correction every 10 ms (cf. NTP [11]) and having to do a rather large adjustment, adding 9 or 11 ms, we can have much smaller changes by adjusting the clock slightly for every data sample and thus approximately $8 \cdot 10^6$ times a second (if the logic analyzer runs at 8 MHz).

F. Post-processing

Each of the testbed's nodes independently collects traces and timestamps these. Yet, as the system is intended to run distributed on several devices, we need to aggregate and process the traces, eventually. As all timestamps are based on the same global clock, we can just collect all timestamped traces and merge these into a common trace. When tracing a GPS 1-PPS (1 pulse per second) signal at one of the testbed nodes, we can use it to stretch or unstretch the recorded time between two GPS time signals for all traces. Thus, we can match our traces to a more accurate timescale (UTC).

G. Implementation

After presenting the design of Grace, we discuss its implementation. We also discuss its integration into our existing testbed [14], to illustrate how Grace can be retrofitted and integrated into existing testbeds.

Synchronization node: For implementing the synchronization node, we use a STM32F401 microcontroller with 86 MHz clock speed and a CC1101 433 MHz radio [15]. We send a timestamp once a second, a choice inspired by the 1-PPS signal GPS receivers generate. The timestamp we send once a second is a 4 byte value and the sole payload of the packet.

CC1101 Packet Format: The total structure of the packet we send after the preamble consists of 2 bytes sync word, one byte each for packet length and address, 4 bytes payload (timestamp) and a 2-byte checksum [15].

Testbed node: As an observer, we use a Raspberry Pi 3B+ in our testbed. For the GPIO tracing, we use an eight-channel logic analyzer featuring a Cypress EZ-USB FX2LP microcontroller [16]. The FX2 consists of an 8051 microcontroller, a USB interface, and i.a. the General Programmable Interface (GPIF). The GPIF allows specifying custom communication protocols via a finite state-machine. It is used to constantly sample data from the Logic Analyzer's input pins into the USB buffer. These components work independently of each other, allowing a deterministic tracing operation without being interrupted by the microcontroller or the USB interface. We build a custom firmware for the FX2, enabling us to focus on tracing sparsely occurring events continuously for the full duration of an experiment. The logic analyzer's state machine samples the state of its (eight) inputs at a frequency of 8 MHz (one sample every 125 ns). Internally, it passes these samples to the USB interface and writes them to the USB buffer. To one pin of the logic analyzer, we connect a radio, the same one mentioned above (CC1101 433 MHz radio). To notify the logic analyzer of a successfully received packet, the CC1101 asserts the successful reception of a reference signal on one of its GPIO pins [15]. We implement the described trace collection and time error correction functionalities and algorithms as part of an application running in user space on the observer (Raspberry Pi). For having a high granularity for the timestamps, we choose a value of 2^{60} as a value for the constant TICKS_PER_SECOND in our algorithms. Moreover, when calculating the new frequency adjustment value, we weight the newest frequency offset estimation with 90% and the previous estimate with 10% (cf. WEIGHT = 0.9). For ease of wiring the logic analyzer and the radio to the Raspberry Pi, and positioning the radio, we design a (non-essential) custom PCB HAT for the Raspberry Pi (see Fig. 1b).

Post-processing: We implement the post-processing to aggregate the GPIO traces into a common human-readable CSV file. Moreover, we also convert it into a VCD file to be able to easily, visually analyze the combined traces with a software like GTKWave.

Cost: As we argue that Grace is a low-cost GPIO tracing system, we present in Table I the cost for the different components, at the time of writing. The table shows, that both

TABLE I: Cost of components for Grace

Component	Cost (€)
CC1101 Radio Module	9
STM32F401 Development Board	9
8-Channel Logic Analyzer	8
Jumper Wires	< 1
Total: Synchronization Node	19
Total: Testbed Node	18

the hardware for a testbed node, as well as the hardware for the synchronization node, stay below 20. Equipping a full testbed of 20 nodes with this GPIO tracing system costs less than 380.

IV. Evaluation

In this section, we experimentally evaluate our time-synchronized GPIO tracing system Grace. We evaluate its degree of time-synchronization and its stability within an actual testbed deployment. We set it into perspective to other time synchronization approaches, discussing its advantages and disadvantages. We evaluate the different sub-systems individually before finally looking at the system as a whole.

A. Evaluation Setup

1) Testbed: We evaluate Grace on our local testbed of 20 nodes (see Fig. 2), spanning the top-most floor of a university building with an area of 500 m^2. The floor was mostly unoccupied during the experiments, yet, as we use 433 MHz for communication, we expect even an occupation to have minimal impact on the evaluation results.

We equip 16 of the 20 nodes, with a logic analyzer and a radio (marked with orange squares in Fig. 2). Additionally, we place the synchronization node in close vicinity of one of the testbed nodes (marked with a red circle in Fig. 2).

2) Metrics: We evaluate Grace in terms of the logic analyzer's frequency stability, the synchronization node's frequency stability, timing offset between testbed nodes, and system-wide time-synchronization stability. We measure frequency deviations and time offsets.

3) Reference Clock: For measuring the exact timing, we require systems with a more accurate clock than the system's clock we evaluate. Therefore, we either use an external logic analyzer (Saleae Logic Pro 8) or the 1-PPS signal output of GPS receivers as a reference clock. The error of our reference clock is up to 50 ns (50 ppm) for the Saleae logic analyzer [17] and up to tens of nanoseconds between two GPS receivers. For the experiments in Sections IV-C and IV-E we use the logic analyzer, as we interface two nodes at once. For the remaining experiments, we use the GPS receivers. The GPS receivers (marked 'G' in Fig. 2) have an external antenna mounted on the outside of the building, to keep the 1-PPS synchronization at the specified accuracy.

Fig. 2: Local testbed of 500 m^2. Red circle: synchronization node; Orange squares: Nodes equipped with our GPIO tracing system (Grace); Blue hexagons: other nodes; Marker G: nodes equipped with GPS.

B. Logic Analyzer Frequency Stability

We start our evaluation by analyzing the frequency stability of a logic analyzer. We, therefore, trace the 1-PPS signal of a GPS receiver with a logic analyzer. In Fig. 3 we show the frequency stability over two hours and the relative occurrence of the different deviations, exemplary for one logic analyzer. This logic analyzer had during the tracing of the GPS signal an average deviation from the nominal frequency of 154.9 ppm. Generally, we expect the frequency stability of the logic analyzers to be within 200 ppm [17]. As 200 ppm equals a time difference of 200 µs within a second, and thus, a maximum time difference of 400 µs between two logic analyzers, this clearly underlines the need for a system with time-synchronized logic analyzers.

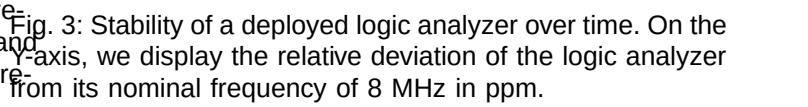


Fig. 3: Stability of a deployed logic analyzer over time. On the Y-axis, we display the relative deviation of the logic analyzer from its nominal frequency of 8 MHz in ppm.

C. Frequency Stability Of Synchronization Node

Next, we investigate the frequency stability of the synchronization node. We configure the synchronization node to send a packet once a second according to its internal clock. With an external logic analyzer, we record the exact sending times over a period of 90 minutes. For that, we record the end of creating a timestamp at the microcontroller by tracing the chip select line. Moreover, we trace the state of the GPIO pin the radio turns off once it is done sending a packet.

Fig. 4a shows the offset of the microcontroller's second from the reference second over the time of the experiment (between 7 and 9 µs). It also shows the resulting offset of the reference time signal from the reference time. Generally, the radio sending times precisely follow the microcontroller with a slight jitter of on average 18.7 ns and a maximum of 78 ns. While the offset between two timestamps is on average 7.1 µs, it is of no concern as the offset will be evenly present on all testbed nodes and thus have at most a minor influence on the distributed time-synchronization. The added jitter in the nanosecond range is no concern for our system's requirements.

After investigating the synchronization node's stability, we next look at the stability of the testbed nodes. Firstly, we

(a) Stability of the synchronization node's signal over 90 minutes. The radio output of the synchronization node follows the microcontroller with a slight jitter of on average 18.7 ns.

(b) Offset distribution of a radio's input signal from the radio output signal of the synchronization node. Avg offset (solid red line): 1.32 µs, Std (dashed red lines): 637 ns.

Fig. 4: Stability of the microcontroller output, the synchronization node's radio output, and the testbed node's radio input.

look at the deviation of the input signal at one testbed node from the output signal of the synchronization node. For that, we interface both nodes with our external logic analyzer simultaneously. On the synchronization node, we trace the GPIO pin, the radio turns off once it is done sending a packet, and on the testbed node, we trace the GPIO pin of the radio that also noti es our time-synchronization system of the availability of a new timestamp. We initially synchronize these two timestamps, to analyze the receiver's variation in offset. Fig. 4b shows an average offset of 1.32 µs with a standard deviation of 637 ns from the synchronization node's time signal.

Next, we look at two nodes in the testbed, close to each other and the signal received by the radios. We once again use the logic analyzer and trace the GPIO pin of the radio that also noti es our time-synchronization system of the availability of a new timestamp. We run several experiments with a total duration of almost 5 hours. When looking at the reception time differences between the two radios, we see the distribution shown in Fig. 5a. The difference between the radio's reception times (without the time correction system) is on average 654 ns (median: 562 ns) with a standard deviation of 487 ns. The maximal measured offset between the two radios is 3.22 µs. 77.4% of the measurements have an offset of less than 1 µs. Even the maximum value of 3.22 µs is suf cient for evaluating the timing of many IoT protocols, including Time-Slotted Channel Hopping (TSCH) [5].

D. Clock Correction

Next, we focus on the full system, including the time error correction. To evaluate this, we include the nodes that have a GPS receiver, and we use the 1-PPS GPS signal traced by the testbed node's logic analyzers. We analyze the time differences of the timestamps associated with the 1-PPS GPS signals. Fig. 5b shows the distribution of the time stamping error of Grace. On average, the system has an error of 1.53 µs with a standard deviation of 644 ns, and a maximum error of 3.75 µs. This clearly shows the advantage of Grace over NTP with a thousandfold higher precision. Moreover, this synchronization is suf cient for analyzing timing in many IoT protocols, including time-critical communication protocols like Time-Slotted Channel Hopping (TSCH) [5] and Chaos [7]. After comparing the system's stability with a GPS reference, we can also compare it to the reference signal our synchronization node sends out. Therefore, we compare the deviation of the local timestamps based on the synchronization signal. Fig. 5c shows similar results to the GPS-based experiment. However, when tracing the synchronization node's time signal, all the offsets between the different receivers get accumulated (cf. Fig. 5a). Overall, the offset, when including all these errors, between any two nodes is on average 2.92 µs with a maximum offset of 7.9 µs.

E. Receiver Stability

From these results, we can conclude that Grace achieves a building-wide time-synchronization in the range of a few microseconds. This does not fully reach the degree of time-synchronization offered by custom solutions with speci c hardware requirements or using FPGA's for the GPIO tracing [9], [10]. However, our system is easily and cost ef ciently retro ttable to existing testbeds and offers a suf cient time synchronization for tracing of many application elds. Moreover, it offers a signi cantly higher degree of clock synchronization than NTP.

V. Related Work

Several testbed architectures presented in the recent years offer GPIO interfacing capabilities. To our knowledge, the rst testbed offering GPIO tracing capabilities is Flocklab [8]. Flocklab's GPIO tracing system directly uses the observer for GPIO tracing and can trace up to 5 GPIO pins at a sampling rate of up to 10 kHz. For time-synchronization, the system uses NTP, reaching a precision of 40 µs. Next to the GPIO tracing, Flocklab also allows GPIO actuation as well as power pro ling. With Tracelab [9], Lim et al. extend Flocklab by a more capable GPIO acquisition system based on an FPGA. They achieve a short-term sampling frequency of up to 100 MHz, and a continuous sampling frequency of around 285 kHz. For time synchronization, they use Glossy on 868 MHz with an FPGA-based clock correction control loop, achieving a maximum time-synchronization error of 1.5 µs.

(a) Occurrences of offsets between two radio receivers. Avg: 0.65 µs, Std: 487 ns.

(b) Offsets between any two traced GPS signals. Avg: 1.53 µs, Std: 644 ns.

(c) Offsets between any two GPIO tracing nodes within our testbed. Avg: 2.92 µs, Std: 864 ns.

Fig. 5: Histograms showing the distribution of offsets between two radio receivers, or between multiple nodes using the full time-error correction system. We show the mean value as a solid red line, and the standard deviation as dashed red lines.

Other testbed architectures like Aveksha [18], Minerva [19], and HATBED [20] use different J-Link tracing methods, including tracing the program counter, or watchpoint tracing in a non-intrusive way. Minerva uses NTP for time synchronization, with precision in the milliseconds range.

A more recent work is Flocklab 2 [10], which uses the programmable real-time unit (PRU) of a Beaglebone Green for GPIO tracing. For time synchronization, the system uses GNSS with an accuracy of approx. 50 ns where available, and the Precision Time Protocol (PTP) with an accuracy of approx. 1 µs at all other locations. Next to the GPIO tracing, it also supports Serial Wire Debug (SWD) tracing through a J-Link debug probe.

Regarding time synchronization, Grace has the highest similarity with Tracelab. However, instead of performing the time synchronization in hardware on an FPGA, we do it at a higher precision in software on a Raspberry Pi when processing the logic analyzer's traces. Regarding the GPIO tracing, we differ from all these solutions in that we use low-cost logic analyzers that do not depend on a specific observer platform.

VI. Conclusion

Testbeds are an important tool for developing and evaluating IoT protocols. While there are many testbeds used by the research community, most of them lack the capabilities to accurately evaluate the timing of time-critical IoT systems.

With Grace, we present an easily retrofittable system capable of tracing the timing of IoT devices by proposing a low-cost GPIO tracing system. Grace uses only low-cost off-the-shelf components, making it retrofittable to existing testbeds for less than €20 per node. We show that Grace can synchronize GPIO events in a testbed with an average time synchronization error of 1.53 µs, and a worst-case time synchronization error of 3.75 µs.

References

[1] A. von See, "Number of Internet of Things (IoT) connected devices worldwide from 2019 to 2030," 2021. [Online]. Available: https://www.statista.com/statistics/1183457/iot-connected-devices-worldwide/

[2] F. Osterlind, A. Dunkels, J. Eriksson, N. Finne, and T. Voigt, "Cross-Level Sensor Network Simulation with COOJA," in IEEE LCN, 2006.

[3] "BabbleSim - A physical layer simulator." [Online]. Available: https://babblesim.github.io/

[4] F. Ferrari, M. Zimmerling, L. Thiele, and O. Saukh, "Efficient network flooding and time synchronization with Glossy," in ACM/IEEE IPSN, 2011, pp. 73–84.

[5] R. F. Heile, R. Alfvin, P. W. Kinney, J. P. K. Gilb, and C. Chaplin, "IEEE Standard for Local and metropolitan area networks–Part 15.4: Low-Rate Wireless Personal Area Networks (LR-WPANs) Amendment 1: MAC sublayer," IEEE, Tech. Rep., 2012.

[6] F. Ferrari, M. Zimmerling, L. Mottola, and L. Thiele, "Low-power wireless bus," in ACM SenSys, 2012, p. 1–14.

[7] O. Landsiedel, F. Ferrari, and M. Zimmerling, "Chaos: Versatile and Efficient All-to-All Data Sharing and In-Network Processing at Scale," in ACM SenSys, 2013, pp. 1–14.

[8] R. Lim, F. Ferrari, M. Zimmerling, C. Walser, P. Sommer, and J. Beutel, "FlockLab: a testbed for distributed, synchronized tracing and profiling of wireless embedded systems," in ACM/IEEE IPSN, ser. IPSN '13, 2013, p. 153–166. [Online]. Available: https://doi.org/10.1145/2461381.2461402

[9] R. Lim, B. Maag, B. Dissler, J. Beutel, and L. Thiele, "A testbed for fine-grained tracing of time sensitive behavior in wireless sensor networks," in IEEE LCN Workshops, 2015.

[10] R. Trüb, R. Da Forno, L. Sigrist, L. Mühlebach, A. Biri, J. Beutel, and L. Thiele, "FlockLab 2: Multi-Modal Testing and Validation for Wireless IoT," in CPS-IoTBench ETH Zurich, 2020.

[11] M. van Steen and A. S. Tanenbaum, Distributed Systems, 3rd ed. distributed-systems.net, 2017.

[12] Electronic Notes, "What is a Logic Analyzer." [Online]. Available: https://www.electronics-notes.com/articles/test-methods/logic-analyzer/basics-tutorial.php

[13] T. Wescott, "PID Without a PhD," in Embedded Systems Programming, 2000. [Online]. Available: https://m.eet.com/media/1112634/f-wescot.pdf

[14] IoT Chalmers, "IoT-Testbed," 2021. [Online]. Available: https://github.com/iot-chalmers/iot-testbed

[15] CC1101 - Low-Power Sub-1 GHz RF Transceiver, Texas Instruments. [Online]. Available: https://www.ti.com/lit/ds/symlink/cc1101.pdf

[16] EZ-USB® Technical Reference Manual, Cypress Semiconductor, Cypress Semiconductor, 198 Champion Court, San Jose, CA 95134-1709, 2019, 001-13670 Rev. *G. [Online]. Available: https://www.infineon.com/dgdl/Infineon-EZ-USB_TECHNICAL_REFERENCE_MANUAL-AdditionalTechnicalInformation-v08_00-EN.pdf

[17] Saleae Support, "Time Measurement Error," 2021. [Online]. Available: https://support.saleae.com/faq/technical-faq/time-measurement-error

[18] M. Tancreti, M. S. Hossain, S. Bagchi, and V. Raghunathan, "Aveksha: a hardware-software approach for non-intrusive tracing and profiling of wireless embedded systems," in ACM SenSys, 2011, p. 288–301.

[19] P. Sommer and B. Kusy, "Minerva: distributed tracing and debugging in wireless sensor networks," in ACM SenSys, 2013, pp. 1 –14.

[20] Y. Li, J. Ma, and T. Zhang, "HATBED: Hardware Assisted Tracing Testbed for Embedded Networked Sensor Systems," in ACM SenSys 2018, p. 327–328.