electronics

MDPI

Article

Python-Based TinyIPFIX in Wireless Sensor Networks

Eryk Schiller *, Ramon Huber and Burkhard Stiller

Communication Systems Group (CSG), Department of Informatics (IfI), University of Zürich (UZH), Binzmühlestrasse 14, CH-8050 Zürich, Switzerland; ramon.huber@protonmail.ch (R.H.); stiller@ifi.uzh.ch (B.S.)
* Correspondence: schiller@ifi.uzh.ch; Tel.: +41-44-635-4337

Abstract: While wireless sensor networks (WSN) offer potential, their limited programmability and energy limitations determine operational challenges. Thus, a TinyIPFIX-based system was designed such that this application layer protocol is now used to exchange data in WSNs efficiently. The new prototype is based on the Espressif ESP32-WROOM-32D Internet-of-Things (IoT) platform, which is becoming famous, as it is inexpensive but powerful compared to older generations of IoT devices. The system implementation is provided in the programming language MicroPython, which provides a simple and efficient implementation, compared to a lower-level programming language. Therefore, this approach focuses on value creation rather than platform-specific implementation difficulties. The system is evaluated in smart home use cases and displays valuable overhead, reliability, and power efficiency. TinyIPFIX outperforms the data overhead of the type–length–value (TLV) paradigm by a factor of 7% when a TinyIPFIX data message carries only two records, and one TinyIPFIX template message is sent per three TinyIPFIX data messages. A further decrease in overhead is observed when the number of data records per message and the number of TinyIPFIX data messages sent per one TinyIPFIX template message increase to larger values. The message delivery between end devices and the application server resides at a very high level, close to 100%, when the transmission reliability is secured with acknowledgments and retransmissions. The energy efficiency resides at the limited level, as the experienced deep sleep power consumption of the ESP32 device resides at the milliwatt level.

Keywords: wireless sensor network (WSN); Internet-of-Things (IoT); TinyIPFIX; Espressif ESP32-WROOM-32D

check for updates

Citation: Schiller, E.; Huber, R.; Stiller, B. Python-Based TinyIPFIX in Wireless Sensor Networks. *Electronics* **2022**, *11*, 472. https://doi.org/10.3390/electronics11030472

Academic Editor: Juan M. Corchado

Received: 22 December 2021
Accepted: 1 February 2022
Published: 5 February 2022

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

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

1. Introduction

Typically, a sensor is a module designed to sense the environment and forward this information to other components, determining a distributed system out of computers or servers. Due to low prices, the deployment of sensors, considered to be constrained Internet-of-Things (IoT) devices, has become ubiquitous. Furthermore, sensors organizing themselves in complex structures, such as networks of a multi-hop nature, are referred to as wireless sensor networks (WSN).

WSNs reflect many beneficial use cases and can support, for example, agriculture, energy, health, smart city, smart home, or supply chain monitoring settings [1]. For example, for the smart home/city and energy area, one can consider energy measurements among each household to plan for precise energy production such that an adequate consumption is backed by suitable production, avoiding energy waste.

With such potentials provided by WSNs, significant challenges also exist, preventing their massive deployment. One such challenge is (a) the limited programmability of sensor devices. Very constrained devices, heavily limited in CPU (central processing unit) cycles, RAM (random access memory), and network capacity, are difficult to program and typically require the knowledge of specialized low-level programming languages. Another challenge is related to the fact that sensors are typically battery powered. This requires them to (b)

run very energy-efficient operations, while frequent replacements of batteries might be impossible if hundreds of devices are deployed in the field for a specific use case.

A three-fold approach is considered in this work to deal with those two limitations: (1) A new generation of sensor devices is selected, which relaxes the constraints put on the CPU, RAM, and networking. (2) A high-level programming language is considered to program energy-efficient operations easily. (3) Optimal protocols are demonstrated to transport information within the environment toward applications.

Thus, the Espressif ESP-WROOM-32D [2,3] module is selected, which offers extended capacity. An ESP-WROOM-32D supports high-level programming languages, such as MicroPython [4] or JavaScript, through an appropriate firmware. The ESP-WROOM-32D device is enriched with the Digi XBee platform [5], which supports a highly constrained and energy-efficient IEEE 802.15.4 network [6]. Furthermore, this work deploys the tiny Internet protocol flow information export (TinyIPFIX) [7], which sends metadata and actual sensor data in separate messages and decreases the communication overhead and energy cost in comparison to other protocols in this domain. As a result, TinyIPFIX enables WSNs' operation to run longer on a single battery charge than regular application protocols, such as message queuing telemetry transport (MQTT) or hypertext transfer protocol (HTTP) [8,9].

The remainder of this paper is structured in the following way. Section 2 introduces those technologies used for this approach as a background that this work builds upon. While Section 3 discusses different design decisions and offers concrete examples of the implementation, Section 4 evaluates the implementation. Finally, Section 5 summarizes the work, draws key conclusions, and outlines future work.

2. Related Work

The Internet protocol flow information export (IPFIX) [10,11] is an application layer protocol with the purpose of sending information about traffic flows in a network. Traffic flow information may be required for administrative purposes, such as billing. IPFIX organizes data into template and data records. This way, redundant meta-information, which is sent in template messages, may be sent less often than the actual data sent in data messages. IPFIX first sends meta information in a template record set and then multiple data record sets that contain very little meta-information. IPFIX is push-based, which means that the sender sends data when they are available. No method allows a receiver to request data; thus, IPFIX messages are only sent in one direction. It implies that there is no way for the sender to recognize whether its sent packet arrived at the destination. The push-based unidirectional traffic is divided into data and template records, which provides IPFIX with data and energy-efficient communication.

2.1. TinyIPFIX

TinyIPFIX [7] is an application layer protocol derived from IPFIX optimized for constrained WSNs. TinyIPFIX has a similar purpose to other protocols in the IoT domain, such as HTTP, Constrained Application Protocol (CoAP), or message queuing telemetry transport (MQTT). TinyIPFIX [8] has less overhead than IPFIX due to the introduction of (1) template and data messages, similar to IPFIX as well as (2) in-network aggregation [12]. There are several implementations of TinyIPFIX [8,13,14] provided in the literature. Ref. [8] evaluates TinyIPFIX implementation [13] in C on MoteLab [15] consisting of 40 TelosB nodes [16]. Ref. [14] provides a Python-based implementation of TinyIPFIX for the Raspberry PI [17] platform and C++ implementation for ESP32 [3] devices implemented in the Arduino integrated development environment (IDE) [18]. However, the low-level implementation of a custom protocol is time-consuming due to difficulties introduced by low-level programming languages. Higher-level languages such as Python do not introduce such problems. Therefore, they are better adapted for value creation.

TinyIPFIX is based on the unidirectional push-based communication paradigm. As with IPFIX, TinyIPFIX splits its messages into template and data messages. Data messages are sent more often than template messages to save energy, while template messages only contain

information on decoding corresponding data messages. This is reasonable because template messages (meta-data) repeat themselves and do not often change like data messages, which in turn carry actual sensor readings. Templates usually do not change on a device. For example, a device, which always sends a 4-byte (i.e., float type) temperature reading, does not suddenly provide an 8-byte (i.e., double type) humidity reading. The reason for the retransmission of template messages is that lost template messages, which are not retransmitted, cause the following data messages to be unusable, as they cannot be decoded.

2.1.1. TinyIPFIX Messages

Figure 1 gives an overview of the TinyIPFIX message structure.

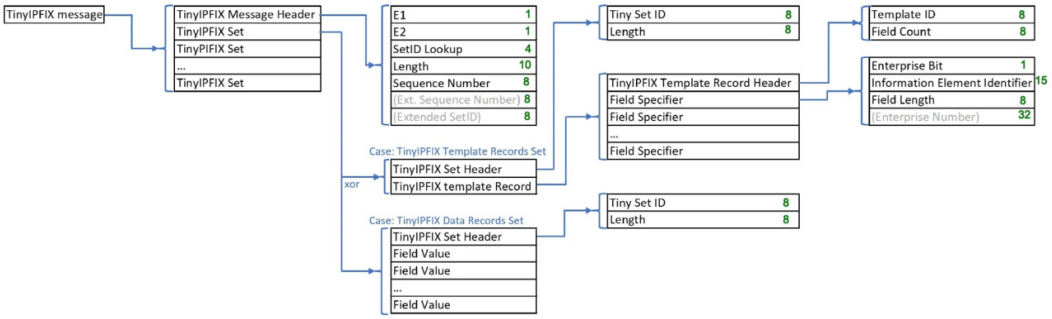


Figure 1. TinyIPFIX Message

The figure can be read in the following way. Each field contains an element of a TinyIPFIX message. When a field is indicated with an arrow leaving this field, it signals an expanded in-depth presentation of that field indicated by that arrow. For example, the leftmost field "TinyIPFIX message" is constructed of a "TinyIPFIX message header" and one (or more) "TinyIPFIX set". Sending a TinyIPFIX message implies sending a TinyIPFIX header and a couple of TinyIPFIX sets. Fields that are not provided with arrows (those with green numbers) are considered atomic. The green numbers indicate how long the fields are in bits. Greyed-out fields are optional and do not have to be present in a TinyIPFIX message. A TinyIPFIX message has one or multiple TinyIPFIX sets, but either all of those sets are template record sets or data record sets. Different TinyIPFIX set types cannot be present in the same message.

2.1.2. TinyIPFIX Message Header

The TinyIPFIX message header is the first part of each TinyIPFIX message. It contains the fields E1, E2, SetID lookup, length, and sequence number. Additionally, it can have either one or both fields, i.e., extended sequence number and extended SetID.

The E1 field gets a value of zero or one. In the case of one, the extended SetID field is present. In turn, the extended SetID field is absent in the case of zero. The E2 field also gets zero or one. Should the value be one, the extended sequence number field is present. Otherwise, it is absent. The field SetID lookup is set to zero in this work, as its definition is not clear in RFC 8272 [7], which specifies TinyIPFIX. The length field stores the total length of the TinyIPFIX message in bytes. The total length includes the length of the TinyIPFIX message header and all the sets in the message.

The field sequence number holds the sequence number of the TinyIPFIX message. The first message that a particular sender sends to a particular receiver has the sequence number 0. Then, the sequence number is increased by one with every packet sent. This way, the receiver knows whether it has received all messages. If one message gets lost, this is logged, but nothing can be done about this fact. If E2 is zero, the sequence number is 8 bits long (i.e., the maximum value of the sequence number is 255). If E2 is one, the sequence number is 16 bits long, which means it can have a maximum value of 65,535. When the maximum value is reached, the sequence number restarts from zero. The extended SetID is never present in this work, assuming E1 is always zero.

Comparing this header with the header of TinyIPFIX messages, it can be noticed that the IPFIX header is 16 bytes long, while the TinyIPFIX header is only 3 bytes long in the case of E1 = E2 = 0. The smaller header is achieved using field compression (e.g., the length field is only 10 bits instead of 16 bits), the two optional fields that can often be omitted, such as the export time. If the export time is needed, it can be sent as a field value in the data record. TinyIPFIX provides a minimum overhead, which can be clearly demonstrated by comparing the IPFIX header with the TinyIPFIX header.

2.1.3. TinyIPFIX Template Records Set

After the header, there are one or more TinyIPFIX sets in a TinyIPFIX message. A message can either contain template record sets or data record sets, but not both simultaneously. Template record sets contain information on how to decode data record sets.

Set Header

A template record set has a set header, which contains a tiny set ID and a length field, as seen in Figure 1. The tiny set ID is always set to two for template record sets, while the length field stores the total length of the template record set in bytes (including the header).

Template Record

After the header, a template record follows. Template records define the structure and meaning of data record sets. The template record is started with a template record header. The template record header contains a template ID and a field count. The template ID is a number between 128 and 255, uniquely identifying the template record (i.e., no two different template records may have the same template ID). Data record sets have a tiny set ID equal to a template ID of a template record. This shows the data record set defined by the template record. The field count stores the number of field specifiers in the template record. The data record sets specified by a template record must also have the same number of field values as the number of field specifiers in the template record.

Field Specifier

A field specifier specifies the format and meaning of a field value from a data record set. A field specifier must specify each field value of a data record set. Each field specifier has an enterprise bit, an information element identifier, a field length, and an optional enterprise number. The enterprise bit is either zero or one. If it gets zero, the enterprise number is not present, and the field specifier is a default field specifier defined by the internet assigned numbers authority (IANA). Using default field specifiers from IANA ensures that they are the same no matter which company uses them and where in the world they are used. If the enterprise bit is one, the enterprise number is present, and the field specifier is a custom one defined internally in a company.

The information element identifier is a unique identifier for an information element. The specification of an information element can then be found using the information element identifier. The specification defines, for example, the data type and description of the information element. This information can then be used to make sense of the field values from a data record set. The field length stores the length of the field value specified by the field specifier in bytes. An enterprise number is a number that is registered with IANA and uniquely identifies the company or organization. Using the information element identifier, the meaning of a field value can be searched on IANA [19]. If the enterprise bit is one, then the information element identifier and the enterprise number can be used to find the specification of the field value inside the company. For this, the company has to prepare a file where they store the specification of the field value.

2.1.4. TinyIPFIX Data Records Set

The TinyIPFIX data record set contains two main elements: Set Header and Field Values.

Set Header

Data record sets also start with a set header. This header contains a tiny set ID and a length field. The tiny set ID is between 128 and 255 (just like the template ID is between 128 and 255) and corresponds to a template ID. The length field stores the length of the set in bytes, including the header.

Field Values

After the header, there follow as many field values as are field specifiers in the template record set that corresponds to the data record set (the template record set with template ID equal to the tiny set ID of the data record set). The meaning (e.g., a temperature value) and data format (e.g., unsigned16) of the field value can then be searched using the field specifier.

2.2. Communication Protocols for IoT Applications

The analysis of underlying communication protocols, which serve as the foundation for TinyIPFIX-based IoT systems, is required to fully understand the benefits of TinyIPFIX. To properly support IoT use cases, several facets, such as communication range, data rates, maximum transmission units (MTUs), communication protocol dependability, and energy efficiency, are required. The IEEE 802.15.4 standard was created specifically for IoT devices in the range of a personal network, i.e., wireless personal area network (WPAN). Prototype IoT applications are often developed and analyzed in the context of low power wide area networks (LPWAN), where Long Range (LoRa) wide area network (WAN), Sigfox, IngenuRPMA, Weightless-N, Long Term Evolution (LTE) machine type communication (MTC), i.e., LTE Cat-M or Narrowband-IoT (NB-IoT), communication technologies are currently under deployment.

A brief overview of IoT communication technologies, based on [9,20,21], is summarized in Table 1. These technologies are characterized in terms of communication range, throughput, and medium access control (MAC) MTU sizes.

Table 1. Performance Comparison

Technology	Communication Range	Throughput	MAC MTU (Byte)
LoRaWAN	2–5 km urban, 15 km suburban	0.3 to 50 kbps	256
SigFox	10 km urban, 50 km suburban	100 bps	Fixed 12
IngenuRPMA	20–65 km	up: 624 kbps down: 156 kbps	64
Weightless-N	5 km urban 30 km suburban	30 kbps to 100 kbps	max. 20
LTE-M	12 km	up: 1 Mbps down: 1 Mbps	1500
NB-IoT	15 km	200 kbps	1600
IEEE 802.15.4	10 m	250 kbps	127

TinyIPFIX is especially of interest for communication technologies supporting very short MTUs, such as LoRaWAN, SigFox, IngenuRPMA, Weightless-N, or IEEE 802.15.4. However, the impact of TinyIPFIX will be limited in cellular applications providing MTUs of 1600 bytes because there is no need to save space in the case of a large MTU. Furthermore, TinyIPFIX is best suited for technologies allowing for in-network aggregation [12,22] like IEEE 802.15.4. Therefore, the advent of the LoRa mesh network [23], which enables in-network aggregation on LoRa concentrators, can have a stimulating impact on the development of TinyIPFIX.

3. TinyIPFIX-based System Architecture

Critical design decisions do impact the resulting architecture. The following specifies the architecture of this work [24].

3.1. TinyIPFIX Sensor Network

Figure 2a provides an overview of the WSN provided, in which end devices measure the environment using a sensor and create TinyIPFIX data packets that contain the sensed data provided as the payload. The packets created are sent either toward a concentrator or directly to the collector. To send, an end device, i.e., ESP32 devices [2,3], passes the data packet to the IEEE 802.15.4 [6] device over the universal asynchronous receiver–transmitter (UART) connection, which in turn sends it using the IEEE 802.15.4 protocol. The packet sent toward a concentrator, or the collector, first arrives at the corresponding IEEE 802.15.4 adapter and is then forwarded over UART to the ESP32-based concentrator or collector node. IEEE 802.15.4 is selected as the transmission protocol because it is a widely used standard for indoor environments and has low-power requirements.

TinyIPFIX packets arriving at a concentrator are aggregated with other packets, and TinyIPFIX messages are derived on the concentrator using their own sensors. Once the desired number of distinct TinyIPFIX packets is aggregated, the packet aggregated is sent to the collector. Moreover, it is also possible to install multiple concentrators from an end device to the collector. To alleviate the problem of packets growing indefinitely, concentrators offer a configurable maximum packet size. If a packet reaches the maximum size, it is then not aggregated further on, but instead, it is directly sent to either the next concentrator or the collector.

As one can observe in Figure 2, transmissions (i.e., solid arrows) are always directed from an end device to a concentrator or from a concentrator, but never the other way around (i.e., from a collector to a concentrator or from a concentrator to an end device). This is because the specification of TinyIPFIX is purely push based and does not implement pull mechanisms to obtain the most recent sensor data immediately. This decision was made to keep TinyIPFIX as energy efficient and straightforward as possible. Finally, TinyIPFIX specifies three kinds of devices within a network.

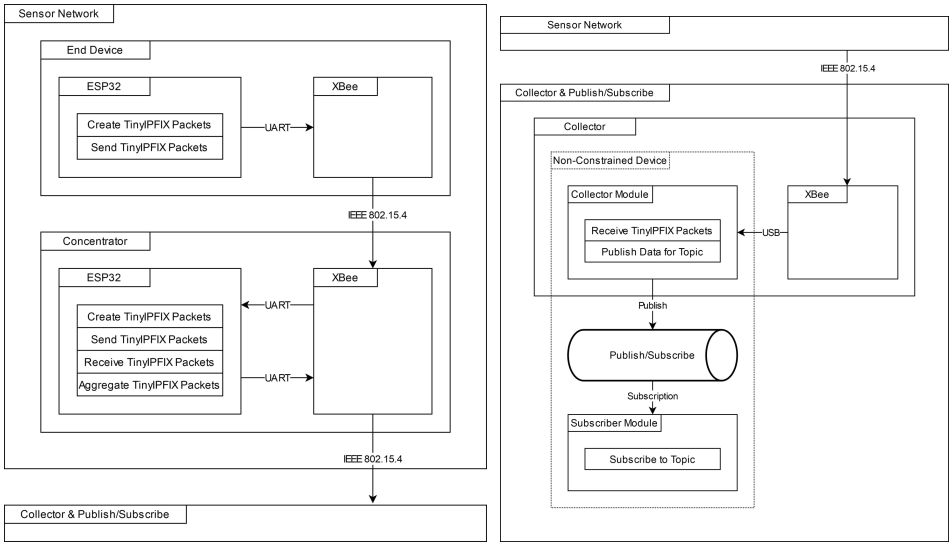


Figure 2. Interactions in the Network. (**a**) Sensor Network. (**b**) Collector and Publish/Subscribe Network

3.1.1. TinyIPFIX Device (End Device)

A TinyIPFIX device is the simplest device. It measures the environment using sensors and sends the data toward TinyIPFIX concentrators or TinyIPFIX collectors. In the TinyIPFIX network, the majority of devices are TinyIPFIX devices.

3.1.2. TinyIPFIX Concentrator

In the TinyIPFIX network, there exist TinyIPFIX concentrators. The concentrators measure the environment, just like TinyIPFIX devices do, but they additionally receive data from other devices. Concentrators aggregate the received data with their own sensed information into a single TinyIPFIX message. This aggregation has the effect that fewer messages, but bigger ones, are sent in the network. Overall, this saves energy because there are more relevant data than overhead in aggregated messages. Another critical task that concentrators satisfy is that they allow multi-hop forwarding if the collector is out of reach. This allows for more extensive networks, while otherwise, the TinyIPFIX device would have to be placed in the range of the collector.

3.1.3. TinyIPFIX Collector

Figure 2b provides an overview of the collector and publish/subscribe (Pub/Sub) broker. While the WSN handles measurements of the environment and ensures that all packets are sent to a central point, the collector and Pub/Sub network handles the processing of the data that arrives from the sensor network. Packets from the sensor network (cf. Figure 2) arrive at the IEEE 802.15.4 interface of the collector. The IEEE 802.15.4 interface passes the packet over universal serial bus (USB) to the non-constrained device that runs the collector module.

Therefore, Figure 2b shows an IEEE 802.15.4 network connected to a non-constrained device, which provisions a collector. All packets of the entire sensor network have to arrive at the collector eventually. The collector module parses these packets received. Furthermore, if a packet containing template records arrives, the unknown template record is stored, while the known ones are ignored. If a packet contains data records, the data are extracted from the packet using known templates already stored. The data are shared with other modules using a Pub/Sub broker, e.g., zero message queue (ZMQ) [25]. Within the Pub/Sub engine, the data are published using the corresponding TinyIPFIX set ID (i.e., SetID) as the Pub/Sub topic. Applications may then subscribe to SetIDs of interest and process the data further. The Pub/Sub engine allows for manageable implementations. An application can access the desired data with just a few lines of code without worrying about how TinyIPFIX works.

3.2. Implementation

To describe a systems approach and its proof of concept, two applications were developed: one just prints all data received at the console on the application server and a second one receives the data subscribed to and stores them in a relational database management system (RDBMS) database, such as MySQL. These cases are supported by the prototypical implementation of the TinyIPFIX protocol and the data processing in detail. The code is available at GitHub [26].

3.2.1. ESP32 Firmware Preparation

The setup of the hardware infrastructure needs to be provided first, such that the microcontroller may be programmed with the desired functionality. This approach used two different ESP32-based boards, i.e., ESP32 DevKitC V4 [27] and SuperB [28], cf. Figure 3a. ESP32 devices may be programmed from a regular PC with USB connectors. The ESP32 DevKitC V4 features a USB port on its own; for the SuperB, the Sparkfun XBee Explorer Dangle is used (i.e., a USB-to-XBee layout connector). Both ESP32-based devices need to be equipped with MicroPython (i.e., appropriate firmware) [4] to be able to parse code written in MicroPython.

At first, the MicroPython firmware has to be downloaded. Then the ESP32-based devices need to be put into download mode to flash them up. The ESP32 DevKitC V4 supports buttons for flashing. To put the ESP32 DevKitC V4 in the download mode, the *boot* button must be pressed, the *EN* button must be pressed and released, and finally, the *boot* button can be released. Putting the SuperB into download mode is more complex because the SuperB does not feature buttons. The *IO0* pin of the device has to be grounded (also referred to as pulled-low or provided with zero), then, while keeping *IO0* grounded, ground is applied to and removed from the *EN* pin, and is finally removed from the *IO0* pin. Once the ESP32 based device is in download mode, the MicroPython firmware may be flashed on the device, which allows the ESP32-based device to be programmed with MicroPython. The MicroPython instructions may be transferred to the ESP32 directly using serial communication or an integrated development environment (IDE) that stores code files on the ESP32 device. MicroPython may parse the code files; the code stored in the *main.py* file is automatically executed upon every boot of the ESP32 device.

3.2.2. Connecting the ESP32-Based Device with XBee

Since UART is used to connect the ESP32-based device with Digi XBee [5], i.e., the IEEE 802.15.4 communication device, cf. Figure 3a, two lines need to be connected between the two devices (i.e., ESP32 and XBee). Additionally, two lines are used to power up the XBee using the ESP32 power and ground pins. If the device is to be used as an end device, two additional lines need to be used to control the XBee awake and sleep periods.

Figure 3a gives an overview of the pins that were used in this work. For a Digi XBee, all marked pins need to be used according to the corresponding ones labeled. For the ESP32-based devices, the pins labeled power and ground also need to be used for ground and power, respectively, but the other pins, i.e., transmit (TX), receive (RX), clear to send (CTS)-XBee, and wake-up-XBee, may also be replaced with other general purpose input output (GPIO) pins. The change of pins needs to be acknowledged in the program code.

To connect the ESP32-based device with XBee, the ESP32 TX (respectively, RX) pin needs to be connected to the XBee RX (respectively, TX) pin. The ESP32-based devices are powered and grounded via USB or a battery. To power the XBee, the ESP32 pins can be used. To this end, the ESP32 power (resp. ground) pin needs to be connected with the XBee power (resp. ground) pin. Should a given device be used as a TinyIPFIX End Device, the wake-up-XBee (resp. CTS-XBee) pin from the ESP32-based device needs to be connected with the sleep-control (resp. CTS) pin of the XBee. If the wake-up-XBee pin transitions from high to low, the XBee wakes up (for this purpose, however, the XBee has to be configured accordingly). When the Xee remains awake and is idle (does not receive data over UART) for a predefined amount of time, it automatically goes to sleep again. When the XBee is awake and ready to send data, the CTS-XBee pin has a low value. This pin is used to check that the XBee is fully awakened before sending data to the XBee device over UART. Once these connections above are established and configured, the ESP32-based device may talk to XBee (and vice versa) over UART. Figure 3b,c shows how the pins of a ESP32 DevKitC V4 and SuperB need to be connected to the pins of an XBee, when those ESP modules are used as an end device.

3.2.3. Configuring XBee Devices

To configure XBee devices, software released by the manufacturer of the XBee—called XCTU [29]—is used. XCTU deployed the IEEE 802.15.4 TH function with the firmware version 2003 on XBee devices. Furthermore, XCTU sends AT (attention) commands to the XBee with the help of a graphical user interface (GUI) while also providing explanations for all options. It is, therefore, easier to use XCTU than to send raw AT commands over the command line. To flash an XBee connected to a computer in XCTU, an appropriate button or CTRL+SHIFT+D can be pressed. When the *update* button is pressed, the XBee function set to be flashed on the device can be chosen.

The XBee network has one personal area network (PAN) coordinator (i.e., IEEE 802.15.4) that manages the network and multiple end devices. The TinyIPFIX collector device is chosen to be the coordinator, while the rest of the XBee devices become IEEE 802.15.4 end devices. The reason for this decision is that the IEEE 802.15.4 coordinator is a single point of failure. Furthermore, the TinyIPFIX collector is a single point of failure as well. Therefore, choosing the TinyIPFIX collector to be the IEEE 802.15.4 PAN coordinator leads to one single point of failure instead of two otherwise, which is also beneficial.

These XBee modules can configure many settings, but particularly interesting for this work are the communication channel (CH), PAN ID (ID), upper 32 bit of the destination address (DH), lower 32 bit of the destination address (DL), 16 bit source address (MY), coordinator enable (CE), MM (MAC mode), defining whether acknowledgments for the transmission are configured, and DIO7 configuration (D7) enabling the CTS flow control.

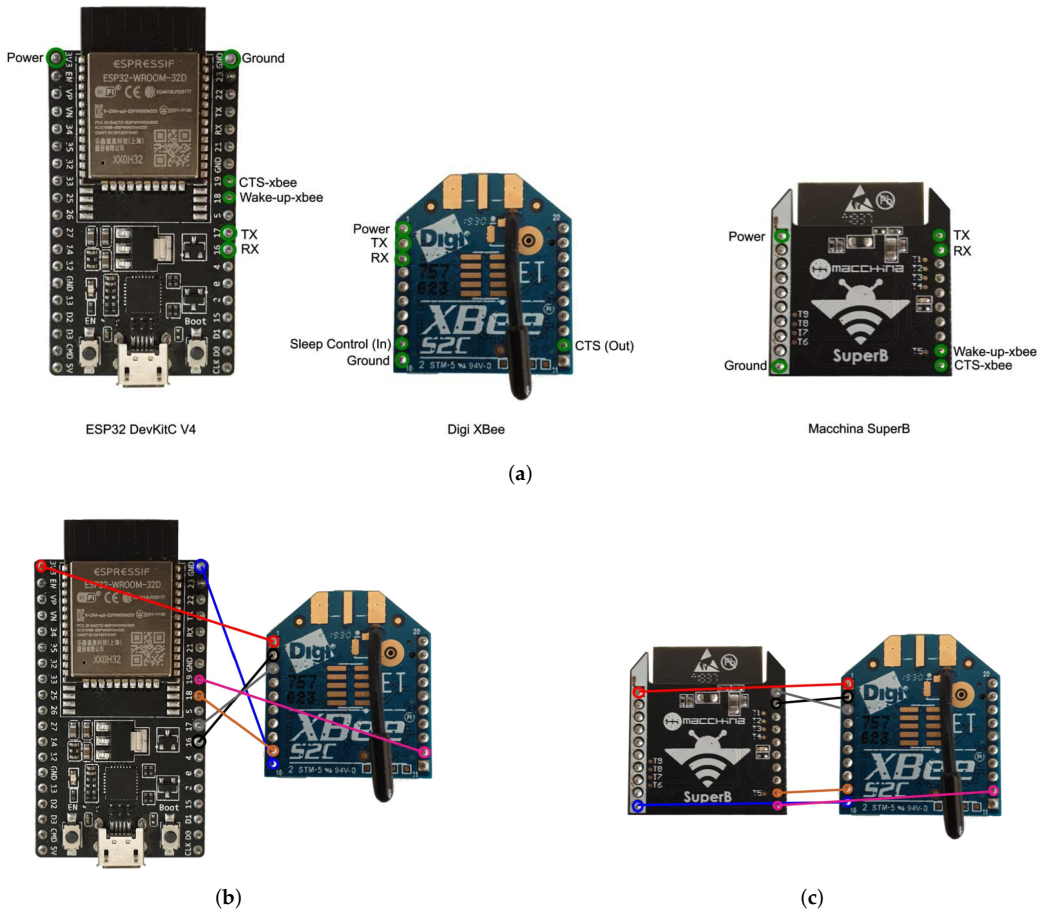


Figure 3. ESP32/Digi XBee Wiring. (**a**) Pin Overview; (**b**) Connecting the ESP32 DevKitC V4 with the XBee (End Device). (**c**) Connecting the SuperB with the XBee (End Device).

3.2.4. TinyIPFIX Protocol Implementation

Figure 4 shows the class diagram that was developed in this work. The figure facilitates the understanding of the implementation details that follow. All classes related to the TinyIPFIX protocol implementation are gathered in the tinyIPFIX.py file on the ESP32 device. For more details on the concrete implementation, the GitHub repository [26] may be consulted.

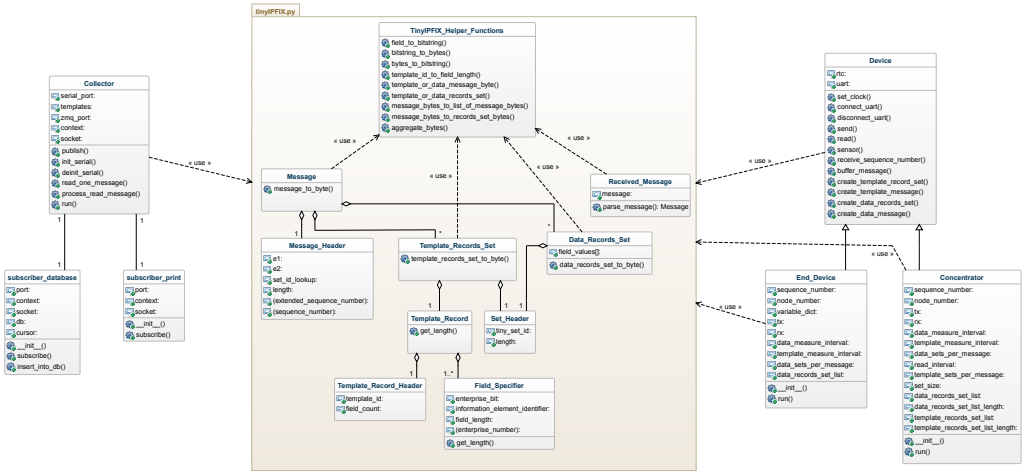


Figure 4. Class Diagram

TinyIPFIX Helper Functions Class

TinyIPFIX helper functions class is the foundation of the TinyIPFIX implementation. The helper class contains useful functions needed for the implementation of other classes. It contains functions allowing for representing TinyIPFIX messages as byte objects (i.e., strings of bytes) and bit strings (i.e., strings of bits). Every TinyIPFIX message may be modified on the byte and bit level. The helper class may convert a bytes object (a string of bytes) into a bit string object (a string of bits) in which distinct bits are manipulated one by one. Finally, the bit string can be converted back into a resulting bytes object. Although the conversion from bytes objects to bit strings seems complex, it allows for simple modifications of bits in comparison to bit-wise operations on bytes objects. Other valuable functions specified relate to TinyIPFIX. Among various methods, there is a function distinguishing between TinyIPFIX data and template messages or a method accepting a bytes object containing an array of TinyIPFIX messages and converting the bytes object into a list of bytes objects, each containing a TinyIPFIX message within.

Tiny IPFIX Message Specific Classes

The remainder of the classes represents different parts of the TinyIPFIX message structure. The generic message class does not specify any distinct attributes, but it offers one function, which is to convert the message object instance to a bytes object that may be, in turn, immediately sent over the IEEE 802.15.4 network. The message may be derived with the help of more specific classes, such as message header, template records set, or data records set classes. The template records set class allows for the derivation of all template records of a template message kept in the message object. The received message class is used to derive the message object from the bytes object received from the network. The received message class accepts a bytes object and returns the corresponding message instance. In such a way, the bytes object can be converted to a message object at the receiver.

Device-Related Classes

The device class offers many functions that are all concerned with creating, sending, and receiving messages and setting the clock or using sensors. It also maintains attributes, which are defined upon the instantiation of the device object. The device class has two main attributes: the real time clock (RTC) for handling timers and UART, which is the object used to send and receive data through peripherals. The TX and RX attributes, maintained by the end device class and the concentrator class, define which pin is used to transmit and receive data over UART. Furthermore, every node maintains a node number, a unique number assigned to each ESP32 device, and a sequence number, an integer increased upon every message sent.

The end device class (for end devices) and the concentrator class (for concentrators) are inherited from the device class. They specify the data measure interval, template measure interval, and data set message attributes. These attributes reflect the data and template measure interval, i.e., the time elapsed between two consecutive data record sets or template record sets sent. On the concentrator, the data measure interval defines the time interval that the concentrator should use to periodically check for new messages to be received from neighboring nodes. The data sets per message attribute defines how many data records a message should contain. If this is set to two, as soon as the device produces two data record sets, it packs them into a TinyIPFIX message and sends them out. The end device does not need any attributes concerning template sets per message, because they always send the same template message with the template record sets they maintain. The concentrator, in turn, maintains template sets per message attribute, because it also receives third-party templates. Due to this fact, the concentrator may send different templates. The concentrator has another attribute called set size, which defines the maximum number of bytes a message may have before it is sent. This ensures that messages do not grow too large due to the aggregation of too many messages. A variable dict attribute from the end device is used to store variables permanently, which is necessary for end devices because they use the deep sleep mode whenever they do not need to send or create messages, and in deep sleep mode, all the content of the non-permanent storage is lost.

For device implementation purposes, a parent class named device and two child classes named concentrator and end device are provided. The device class handles the reading of sensor values, the connection to the XBee (reading and sending data), and the creation of TinyIPFIX messages using the sensor values. The device class implements two functions to create the template record set and message. Those functions are responsible for creating the template message that contains meta-information about data messages. Similarly, the functions create data record's set and create data message are responsible for the creation of data messages. Those functions instantiate a TinyIPFIX message object using different classes that help to derive various components of the message in the protocol.

The class variables' data measure interval, template measure interval, and read interval hold values on the elapsed time between two data measures, two template creations, and two UART reads. The reading interval should not be too large to receive all packets sent from buffers. The value depends on how many devices send data to the concentrator.

Regarding timers (cf. the RTC attribute), data_i, template_i, and read_i hold a timer to the next data measure, template creation, or UART read. Initially, template_i is set to zero; this ensures that a template is sent before data are sent and that the data may be always decoded at the receiver. If data_i, template_i, and read_i are greater than zero, the device does not need to do anything and is set to sleep until one of the timers expires. When the timer expires (i.e., reaches 0), the device needs to perform a given action (e.g., perform a UART read).

When a measurement is performed, the device checks whether it needs to send the message or whether it can still perform other measurements. This is configured in variables containing the maximum message size and maximum sets length. Should the data message be sent and the template record set list is not empty, a template message is sent first containing all template records sets that are currently in the template record set list. This ensures that newer data are never sent before the corresponding template (i.e., containing the prescription on decoding the data). The implementation of the run function on the end device class is similar to the run function from the concentrator class. The fundamental difference, however, is that the end device only creates and sends messages and does not read and aggregate messages. Furthermore, the end device class has a *deep sleep* functionality, which requires some parameters to be stored on flash permanently, even if the device is in the deep sleep mode (i.e., powered off).

Implementation of Collector and Application Related Classes

Three classes are created and named collector, subscriber database, and subscriber print. As noted earlier, these three classes are implemented using Python as a programming language, while all other classes depend on MicroPython. The collector class has a list of templates as an attribute that stores all templates known to the collector. It has a serial port that is used to talk to the XBee connected. Furthermore, it offers a ZMQ port, a context, and a socket which are used to publish data. To publish data, it uses the publish function that uses the above-mentioned attributes. The function 'read one message' reads a message using the serial port, and the function process read message then publishes the data if a data message is read, or adds a template to the known templates if it is a template message containing a yet unknown template. While an application database class subscribes to messages published from the collector class and then stores them in a MySQL database, the application print class subscribes to messages published from the collector class and then prints them on the console.

The collector class receives and decodes all messages from the entire network. For the decoding of data messages, it keeps a list of template messages stored in the cache. Then, it publishes the messages using the SetID as the topic used by the broker (ZMQ broker topic). Finally, the two application classes, i.e., Subscriber_Database and Subscriber_Print, subscribe to the topic of interest, allowing them to receive messages based on their subscription. The Subscriber_Database application stores all data received in a database, while the Subscriber_Print class application prints all data received at the console.

4. Evaluation

To perform a realistic system's evaluation, devices are placed within a home setup providing an IEEE 802.15.4 connected structure. There are seven devices in total, i.e., one collector, two concentrators, and four end devices.

4.1. Network Configuration

All devices are set the same CH, ID, and DH to values of 26, 7,385, and 0, respectively (cf. Section 3.2.3). The value of DL is variable on every device. When DH is set to zero and DL is set to a value smaller than 0xFFFF, the IEEE 802.15.4 16-bit address resolution is used for transmission (i.e., instead of 64 bit address resolution). However, the IEEE 802.15.4 65536 (0xFFFF) address resolution is still perfectly acceptable in smart home application scenarios. MY is a variable 16 bit source address, CE is sent to the *coordinator mode* on the PAN coordinator (i.e., TinyIPFIX collector) or in *end device mode* on all other devices. The collector and the concentrator set SM equal to *no sleep*, while end devices are set to *cyclic sleep with a pin wakeup*. In this mode, the XBee goes to sleep when it is idle for a time and can be woken up using a pin. D7 is set to *disabled* for the collector and concentrators, and set to *CTS flow control* for end devices. If set to *CTS flow control*, then the XBee CTS pin can be used to find out whether the XBee is ready to send data.

Table 2 shows the MY (source) addresses and DL (destination low) addresses of each XBee. The DL address of the collector XBee is a randomly chosen unused address because the coordinator never sends anything. Additionally, the MY address of the end devices is not important because it never receives anything. In future work, a pull mechanism can be implemented, in which case the MY address of end devices becomes important and the DL address of the collector XBee can be set as well. All devices form a tree structure, with end devices (i.e., A.1, A.2, B.1., and B2), concentrators (i.e., A and B), and collector being the leaves, branches, and the root, respectively, using the simple static tree routing implemented for packet forwarding. One can see in Table 2 that End Device A.1 forwards packets to the address A00A, which is Concentrator A. Concentrator A, in turn, forwards its packets to C001, which is the collector in this network.

Table 2. XBee MY and DL Addresses

Device Name	MY	DL
Collector	C001	E99E
End Device A.1	E0A1	A00A
End Device A.2	E0A2	A00A
End Device B.1	E0B1	A00B
End Device B.2	E0B2	A00B
Concentrator A	A00A	C001
Concentrator B	A00B	C001

4.2. Sensor Configuration

One temperature record set is defined and transported between all devices (i.e., concentrators and end devices) and the application server. The record consists of a hardcoded node identifier (2 bytes), temperature readout (4 bytes), and textual timestamp (19 bytes). The temperature value is provided by the hall temperature sensor residing on the ESP32 chip. The hall sensor returns an elevated temperature compared to the current room temperature. At the time of writing this article, the hall sensor provided the authors with around 27 °C, when the experienced room temperature was around 23 °C. However, ESP32 is compatible with many sensors (e.g., based on UART connectors or providing a signal as voltage) attached to the input/output pins of the device. Finally, the timestamp is derived from the RTC attribute, which exposes a daytime function. The daytime returned is converted into the textual timestamp of 19 bytes on the sensor.

4.3. Data Overhead

The overhead of transmission in TinyIPFIX is now compared against the overhead of a simple type–length–value (TLV) approach [8,13]. In a TLV data representation, a message is constructed by combining the type, length, and value elements within the message fields. A message, carrying three parameters measured in the environment, will carry type_1, length_1, value_1; type_2, length_2, value_2; and type_3, length_3, value_3. Typically, type and length fields have a fixed length, while the value field has a variable length defined by the length field. To provide a fair comparison, the TLV system of a similar resolution is compared against TinyIPFIX. TinyIPFIX can support 2^{15} internet assigned numbers authority (IANA) predefined information elements. Additionally, 2^{32} enterprises, identified by the enterprise number, may define another 2^{16} information elements. Under the assumption that only one enterprise number is provided within a network, TinyIPFIX could support 2^{16} different information elements, which corresponds to 2^{16} different types that have to be encoded within a 2-byte type field. Furthermore, the length field is assumed to be 1 byte long, which corresponds to a regular size of the length field in TinyIPFIX. In TLV, the relative overhead is calculated as $O = (\sum_i \text{len}(type_i) + \sum_i \text{len}(length_i)) / \sum_i \text{len}(value_i)$, where the len(·) function provides the length of the corresponding field in the message. Only the value field is the actual payload, while type and length fields are the message overhead.

Considering an assumption that one record consists of a 2-byte sensor identifier, a 4-byte float sensor value (e.g., a temperature readout), and a timestamp, which equals 19 bytes that correspond to a textual timestamp used by MySQL [30], a TLV packet would maintain a 3-byte overhead per data item (i.e., 2 bytes for the type field and 1 byte for the length field) accompanying all items, i.e., the 2-byte sensor identifier, the 4-byte temperature readout, and the 19-byte timestamp. In total, 9 bytes of overhead and 25 bytes of data are sent in such a record, which results in 36% overhead per record.

The total overhead displayed by TinyIPFIX comprises the overhead introduced by TinyIPFIX template messages and data overhead provided within TinyIPFIX data messages. In this example, a TinyIPFIX template message has to maintain one template definition,

in which the enterprise number is provided. However, the TinyIPFIX-specific extended sequence number and extended SetID may be abandoned (cf. [7]). Additionally, a template message provides 7-byte fixed overhead as well as an additional 7-byte overhead for every field specifier. In this setting, a TinyIPFIX template message shows a 28-byte overhead i.e., when a TinyIPFIX template message is sent with three field specifiers in one record. Furthermore, a data message contains a fixed 5-byte header, a 2-byte set header, and a variable amount of data per set. Therefore, the total overhead of TinyIPFIX is dependent on the frequency of template messages, i.e., how many data messages are sent per template message, and the number of data records per data message sent. Figure 5 shows the overhead of TinyIPFIX compared to the TLV approach, depending on these parameters mentioned above.

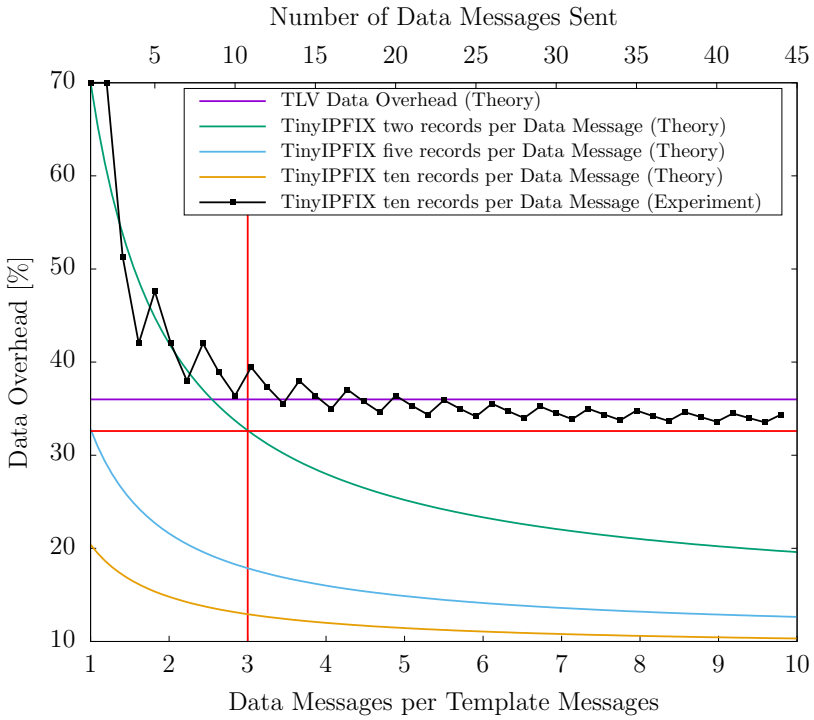


Figure 5. TinyIPFIX and TLV Overhead Comparison

As demonstrated in Figure 5, TinyIPFIX may, in specific settings, outperform the TLV reporting method. TinyIPFIX assumes that template messages are re-sent after many TinyIPFIX data record sets. In the worst case, the delayed template message may lead to a loss of several data packets, while the sink is unable to decode them immediately. However, typically, non-decoded data packets remain in a cache on the collector and wait for the corresponding template messages to come. Finally, the item cached may become successfully decoded once the corresponding TinyIPFIX template packet arrives. Figure 5 displays the constant overhead of 36% for the TLV method represented with a violet horizontal line. Furthermore, the variable data overhead of TinyIPFix is provided for the various number of records stored in a data message and the varying frequency of data messages per templates messages sent (i.e., the bottom x-axis), indicated with the green, blue, and orange lines.

The black line represents an experiment in which only Collector and Concentrator A are activated in the network. The concentrator is configured to send its identifier (i.e., 2 bytes), a temperature readout (i.e., 4 bytes), and a timestamp (i.e., 19 bytes). The number of data messages per template message is configured at 3 (i.e., the concentrator sends three data messages per one template message), while the number of value fields is configured at two (i.e., there are two measurements gathered in a data message). It is worth noting that only data values count as the payload, while other TinyIPFIX-related fields (e.g., headers)

count as the overhead. The relative data overhead of the TinyIPFIX scheme is calculated. To obtain the relative data overhead, the absolute overhead (i.e., in bytes) is divided by the size of the data payload sent. The black line displays the relative overhead as a function of data messages sent (i.e., top x-axis). As the experiment holds two field values and uses the three data messages per template message transmission scheme, a red vertical line is drawn at $x = 3$, according to the bottom x-axis, i.e., three data messages per template message that crosses the green line (i.e., TinyIPFIX holding two records per data message). Then, another red horizontal line is drawn, which connects the aforementioned crossing with the y-axis. This line displays the asymptotic performance of the TinyIPFIX holding two items in a data message, where three data messages are issued per template message. It is well observed that the black line reflecting a real experiment asymptotically approaches the horizontal red line when the number of data messages sent grows along the top x-axis. Furthermore, this TinyIPFIX scheme (i.e., black line) suffers an asymptotic overhead of around 33.5% overhead. Thus, it already outperforms the TLV constant overhead of 36%, i.e., by a factor of around 7%.

However, TinyIPFIX data messages provide two or more data field values at a time. In the case that 10 data field values per TinyIPFIX data message are sent, while a TinyIPFIX template repeats every 10 consecutive TinyIPFIX data messages, the data overhead decreases to around 10%.

4.4. Transmission Reliability

To measure the transmission reliability (i.e., how many data record sets sent by end devices or concentrators arrive at the collector), the test network (cf. Section 4.1) runs for one hour. This work measures the number of data record sets arriving at the collector and concentrators as the function of packets originating at end devices.

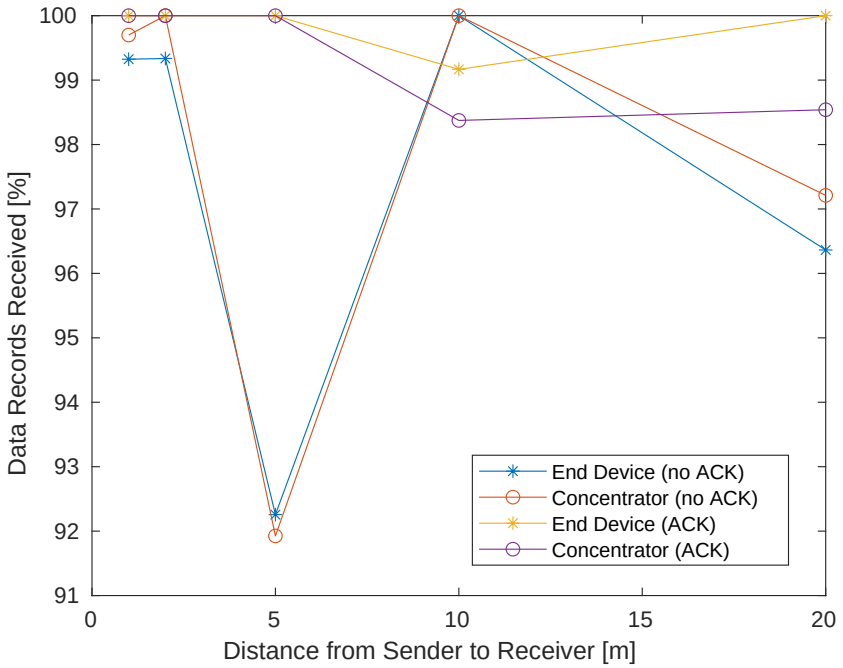


Figure 6. Transmission Reliability

Multiple measurements were performed, where two parameters were altered between measurements. The first parameter is the distance between devices. Initially, these devices spanned a distance of 1 m, i.e., the distance between end devices and the concentrators, and the distance from the concentrators to the collector was set to 1 m. Then, further measurements were performed, where the distance between devices was set to 2, 5, 10, and 20 m. The second parameter was the MM (cf. Section 3.2.3), which controls the IEEE 802.15.4 acknowledgment. All devices in the WSN were configured to create a data record

set every 10 s and send a data message as soon as two data record sets were created. Figure 6 shows the results of these measurements.

For every setup, it was measured how many data record sets arrived at the collector and how many were lost during an hour. For the distances of 10 and 20 meters, it was additionally measured how many data record sets arrived when the XBees was configured to use retransmissions and acknowledgments, i.e., re-send a message when an acknowledgment corresponding to a given message did not arrive.

Overall, it can be observed that most of these data record sets sent successfully arrived at the destination. The only measurement providing quite disappointing reliability was at a distance of 5 m. Unfortunately, no clear explanation could be found. One possible explanation could be that some other device was sending on a similar frequency at that very moment. It can also be observed that data messages sent by the concentrator were received more reliably, which was to be expected because data messages of end devices have to be sent twice (once by the end device and once by the concentrator) until they reach the collector. Furthermore, it can be observed that placing devices at longer distances does not significantly reduce the transmission reliability when acknowledgments are used. These measurements also confirm that the implementation of TinyIPFIX was successful and reliable. All data record sets were successfully created, and all data items received were decoded and stored in the database. The concentrator works in the desired way and aggregates the defined number of messages into one message, ensuring that a data message is never sent before the template message.

4.5. Energy Consumption

A Drok UM25(C) USB power meter [31] was used to measure the energy consumption of the devices in different configurations. The power meter plugs into a USB port (e.g., a PC) which powers the meter. Then, a measuring device is connected to the power meter also over USB (i.e., on the other end). Finally, the power meter shows how much power is used in milliwatts (mW) and over which period.

Table 3 outlines the power consumption of different devices in different states. In the table, ESP32 refers to the ESP32 DevKitC V4, while SuperB refers to the Macchina SuperB. When end devices are in the deep sleep mode, the XBee of the end device automatically enters its sleep mode. SuperB uses less energy than the ESP32 DevKitC V4. End devices save lots of energy compared to concentrators because they go into the deep sleep mode and put the XBee to sleep mode as well. For concentrators (currently, grid powered), the deep sleep mode is not currently implemented because they do not only need to send messages, but they also need to receive them, which is not possible while being in the deep sleep mode.

Table 3. Energy Consumption based on Device Type and State

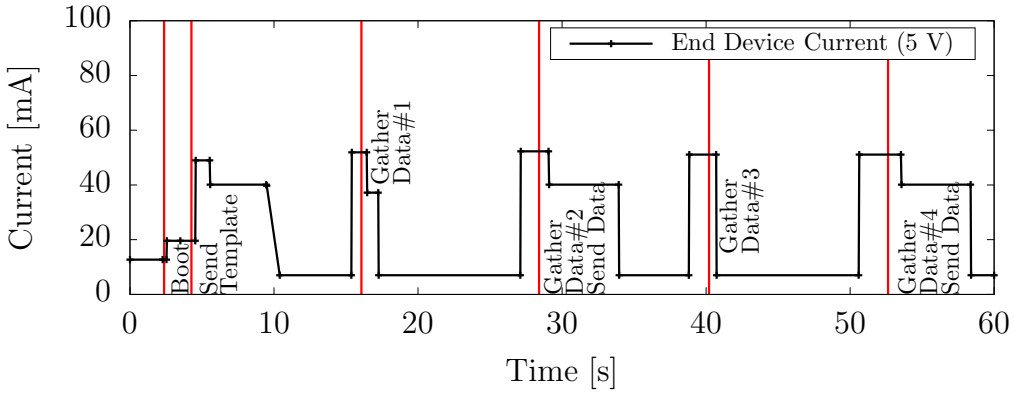
	Deep Sleep	**Idle**	**Sending**	**Receiving**
ESP32 End Device	35 mW	190 mW	344 mW	-
SuperB End Device	20 mW	158 mW	308 mW	-
ESP32 Concentrator	-	328 mW	334 mW	390 mW
ESP32 without XBee	35 mW	192 mW	-	-
XBee without ESP32	-	168 mW	-	168 mW

Figure 7 shows the ESP32-based end device and concentrator current. In this setup, the end device is configured to send measurements roughly every 10 s. It is visible that in the deep sleep mode (i.e., between readings), the device consumes around 35 mW, while the peaks are at 260 mW.

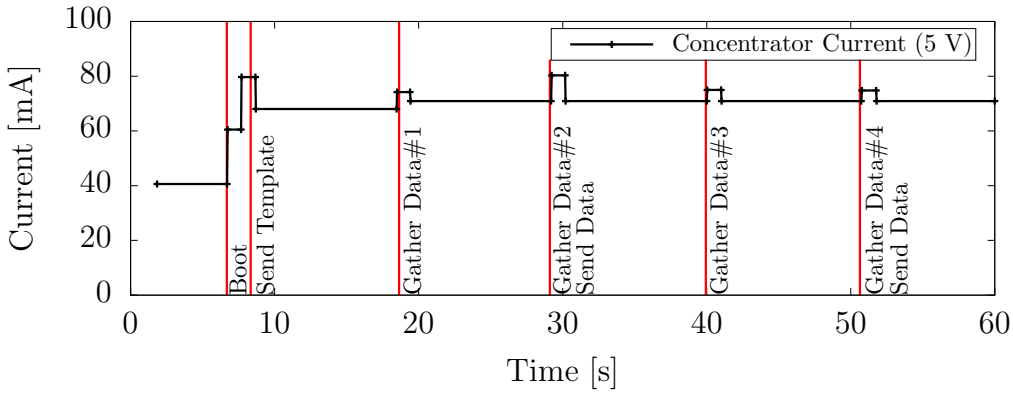
The current consumed by Concentrator A is measured when no end devices are attached to it. The concentrator works by consuming 355–400 mW power. As the power is

measured in a dynamic state, the power values might differ between this experiment and the previous experiment, in which the measured setup was static, i.e., residing in a given state for a longer duration (cf. Table 3).

Figure 8 shows the energy consumption of different devices as a function of messages sent per hour. One can observe that sending more messages greatly increases the energy consumption for end devices, while it does not matter in the case of concentrators. This has to do with the fact that end devices are in the deep sleep mode most of the time, while concentrators are only in idle mode such that they do not lose any arriving messages. The energy consumption mainly depends on the time that devices are in the sleep or deep sleep mode. Using as a battery an alkaline battery of 4200 mWh, the SuperB device reporting one data record per hour could last for around 9 days. Thus, the implementation of TinyIPFIX in these settings is considered a success; however, the currently configured energy expenses of ESP32 devices are still high due to elevated deep sleep power consumption of 20 mWh.



(**a**) End Device Current



(**b**) Concentrator Current

Figure 7. Current Measurements for an End Device and a Concentrator

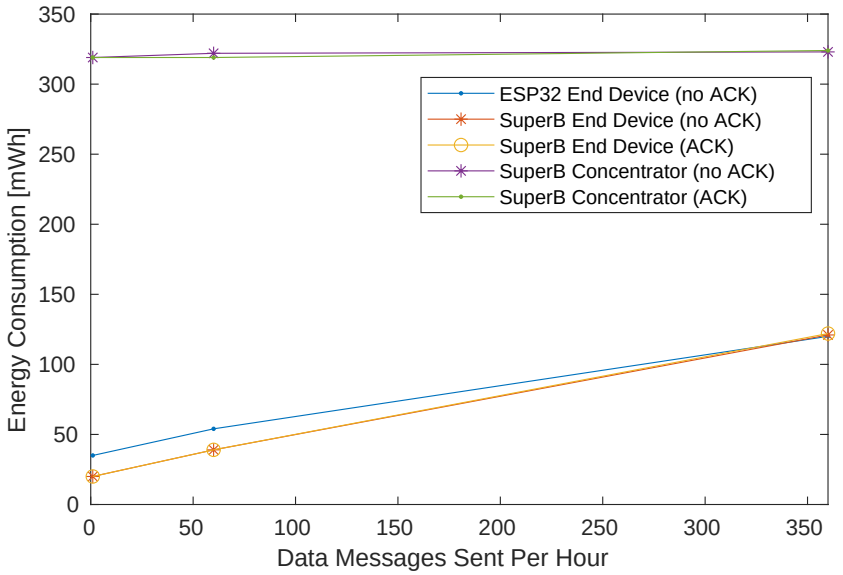


Figure 8. Energy Consumption based on Message Sending Interval

5. Summary, Conclusions, and Future Work

This work implemented a TinyIPFIX platform using Espressif ESP-WROOM-32D devices. TinyIPFIX was selected as a wireless sensor network (WSN) data transport mechanism. As soon as data arrives at the sink, TinyIPFIX messages are provided to the Pub/Sub engine, implemented with the help of the ZMQ message broker. Finally, two applications using the message broker were implemented.

Two ESP-WROOM-32D devices were chosen as the hardware platform: Espressif ESP32 DevKitC V4 and Macchina SuperB. These devices were pre-paired for programming using MicroPython. Then, each ESP device was equipped with a Digi XBee board, which features the IEEE 802.15.4 standard, allowing for low power communication among devices in the WSN. Furthermore, all components of the network were implemented with the help of MicroPython (i.e., end devices and concentrators) or Python (i.e., the collector). Finally, the energy consumption of those devices running TinyIPFIX was evaluated successfully, since the primary method of reducing the energy consumption in the WSN is to leave devices as long as possible in the deep sleep mode (i.e., ESP32 and XBee devices).

In conclusion, TinyIPFIX maintains a more negligible data overhead in specific application scenarios than regular type–length–value (TLV) data transfer. Furthermore, it was demonstrated experimentally that the Python-based TinyIPFIX works well in a home-based IEEE 802.15.4 network, providing almost a 100% delivery ratio. Thus, this solution offers a uniform Python-based implementation spanning multiple elements of the system, including TinyIPFIX end devices, concentrators, and the collector, as well as the ZMQ broker and applications residing on the application server (i.e., the collector). The Python-based environment provides much faster value creation than older systems do, depending on low-level programming languages.

In the near future, different improvements will be considered. A time synchronization-based solution will allow concentrators to enter the deep sleep mode without losing any data packet coming from end devices. As a result, this will reduce the energy consumed by concentrators. Additionally, the elevated deep sleep current currently experienced on ESP32 devices (e.g., 20 mW on the SuperB device in milliamperes) has to be decreased, allowing for an extensive life span of the network.

Author Contributions: Conceptualization, E.S.; methodology, E.S.; software, R.H.; validation, E.S. and R.H.; formal analysis, E.S. and R.H.; investigation, E.S., R.H., and B.S.; resources, B.S.; data curation, E.S. and R.H.; writing—original draft preparation, R.H.; writing—review and editing, E.S., R.H., and B.S.; visualization, E.S and R.H.; supervision, E.S. and B.S.; project administration, B.S.; funding acquisition, B.S. All authors have read and agreed to the published version of the manuscript.

Funding: This research was partially funded by (a) the University of Zürich and (b) the European Union's Horizon 2020 Research and Innovation Program under Grant Agreement No. 830927, the CONCORDIA project.

Data Availability Statement: The software generated in this work is publicly available on GitHub https://github.com/ramonhuber/TinyIPFIX-for-ESP32, accessed on 1 December 2021.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Akyildiz, I.F.; Su, W.; Sankarasubramaniam, Y.; Cayirci, E. Wireless Sensor Networks: A Survey. *Comput. Netw.* **2002**, *38*, 393–422. [CrossRef]
2. Espressif Systems. *ESP32-WROOM-32D & ESP32-WROOM-32U Datasheet*, V1.9. 2019. Available online: https://www.espressif.com/sites/default/files/documentation/esp32-wroom-32d_esp32-wroom-32u_datasheet_en.pdf (accessed on 30 May 2001).
3. Maier, A.; Sharp, A.; Vagapov, Y. Comparative Analysis and Practical Implementation of the ESP32 Microcontroller Module for the Internet of Things. In Proceedings of the Internet Technologies and Applications (ITA), Wrexham, UK, 12–15 September 2017; pp. 143–148.
4. MicroPython Homepage. Available online: https://micropython.org/ (accessed on 4 September 2020).
5. Digi International. *XBee/XBee-PRO S2C Zigbee RF Module User Guide*, AG. 2020. Available online: https://www.digi.com/resources/documentation/digidocs/pdfs/90002002.pdf (accessed on 1 December 2021).
6. IEEE Standard for Low-Rate Wireless Networks. *IEEE Std 802.15.4-2020 (Revision of IEEE Std 802.15.4-2015)*. 2020. Available online: https://ieeexplore.ieee.org/document/9144691 (accessed on 1 December 2021).
7. Schmitt, C.; Stiller, B.; Trammell, B. TinyIPFIX for Smart Meters in Constrained Networks. Available online: https://datatracker.ietf.org/doc/html/rfc8272 (accessed on 28 November 2018).
8. Schmitt, C.; Kothmayr, T.; Ertl, B.; Hu, W.; Braun, L.; Carle, G. TinyIPFIX: An Efficient Application Protocol for Data Exchange in Cyber Physical Systems. *Comput. Commun.* **2016**, *74*, 63–76. [CrossRef]
9. Stiller, B.; Schiller, E.; Schmitt, C. An Overview of Network Communication Technologies for IoT. In *Handbook of Internet-of-Things*; Ziegler, S., James, M., Eds.; Springer: Cham, Switzerland, 2020; Chapter 12.
10. Claise, B.; Trammell, B. Information Model for IP Flow Information Export (IPFIX). Available online: https://datatracker.ietf.org/doc/html/rfc7012 (accessed on 1 December 2021).
11. Claise, B.; Trammell, B.; Aitken, P. Specification of the IP Flow Information Export (IPFIX) Protocol for the Exchange of Flow Information. Available online: https://datatracker.ietf.org/doc/html/rfc7011 (accessed on 1 December 2021).
12. Krishnamachari, L.; Estrin, D.; Wicker, S. The Impact of Data Aggregation in Wireless Sensor Networks. In Proceedings of the 22nd International Conference on Distributed Computing Systems Workshops, Vienna, Austria, 2–5 July 2002; pp. 575–578.
13. Kothmayr, T. Data Collection in Wireless Sensor Networks for Autonomic Home Networking. Bachelor Thesis, Department of Computer Science, Technische University of Munich, Munich, Germany, 2010.
14. Petija, R.; Glevaňák, M.; Kucan, M.; Fecil'ak, P.; Jakab, F. Experimental Implementation of TinyIPFIX Protocol for Arduino and Raspberry Pi Platform. In Proceedings of the 18th International Conference on Emerging eLearning Technologies and Applications (ICETA), Košice, Slovenia, 12–13 November 2020; pp. 519–524.
15. Werner-Allen, G.; Swieskowski, P.; Welsh, M. MoteLab: A Wireless Sensor Network Testbed. In Proceedings of the Fourth International Symposium on Information Processing in Sensor Networks (IPSN'05), Los Angeles, CA, USA, 24–27 April 2005; pp. 483–488. [CrossRef]
16. Polastre, J.; Szewczyk, R.; Culler, D. Telos: Enabling Ultra-Low Power Wireless Research. In Proceedings of the Fourth IEEE International Symposium on Information Processing in Sensor Networks (IPSN), Boise, ID, USA, 15 April 2005; pp. 364–369.
17. Raspberry Pi Foundation Group. Raspberry Pi Products. Available online: https://www.raspberrypi.com/products (accessed on 2 October 2021).
18. Louis, L. Working Principle of Arduino and Using It as a Tool for Study and Research. *Int. J. Control. Autom. Commun. Syst. (IJCACS)* **2016**, *1*, 21–29. [CrossRef]
19. IANA IP Flow Information Export (IPFIX) Entities. Available online: https://www.iana.org/assignments/ipfix/ipfix.xhtml (accessed on 5 September 2020).
20. Minaburo, A.; Pelov, A.; Toutain, L. LP-WAN Gap Analysis. Available online: https://tools.ietf.org/html/draft-minaburo-lp-wan-gap-analysis-00 (accessed on 28 November 2018).
21. Raza, U.; Kulkarni, P.; Sooriyabandara, M. Low Power Wide Area Networks: An Overview. *IEEE Commun. Surv. Tutor.* **2017**, *19*, 855–873. [CrossRef]
22. Sgier, L. TinyIPFIX Aggregation in Contiki. Available online: https://files.ifi.uzh.ch/CSG/staff/schmitt/Extern/Theses/Livio-Sgier-Internship.pdf (accessed on 31 May 2021).
23. Ebi, C.; Schaltegger, F.; Rüst, A.; Blumensaat, F. Synchronous LoRa Mesh Network to Monitor Processes in Underground Infrastructure. *IEEE Access* **2019**, *7*, 57663–57677. [CrossRef]
24. Schiller, E.; Huber, R.; Stiller, B. Python-Based TinyIPFIX in Wireless Sensor Networks. In Proceedings of the 46th IEEE Conference on Local Computer Networks (LCN 2021), Edmonton, AB, Canada, 4–7 October 2021; pp. 431–434. [CrossRef]

25. ZeroMQ Messaging Patterns - Publish/Subscribe. Available online: https://learning-0mq-with-pyzmq.readthedocs.io/en/latest/pyzmq/patterns/pubsub.html (accessed on 10 October 2020).
26. Huber, R. TinyIPFIX for ESP32, GitHub Repository. Available online: https://github.com/ramonhuber/TinyIPFIX-for-ESP32 (accessed on 1 December 2021).
27. ESP32-DevKitC V4 Getting Started Guide. Available online: https://docs.espressif.com/projects/esp-idf/en/latest/esp32/hw-reference/esp32/get-started-devkitc.html (accessed on 7 November 2020).
28. Macchina LLC. SuperB ESP32 Breakout—Overview. Available online: https://docs.macchina.cc/superb-docs/hardware (accessed on 20 September 2020).
29. XCTU. Available online: https://www.digi.com/products/embedded-systems/digi-xbee/digi-xbee-tools/xctu (accessed on 7 November 2020).
30. MySQL Data Type Storage Requirements. Available online: https://dev.mysql.com/doc/refman/5.6/en/storage-requirements.html#data-types-storage-reqs-date-time (accessed on 29 September 2020).
31. DROK. UM25(C) Power Meter. Available online: https://www.droking.com/USB-C-Power-Meter-LCD-Display-USB-Meter-DC-4-24V-5A-UM25-Type-C-Voltage-and-Current-Tester (accessed on 1 December 2021).