Python-Based TinyIPFIX in Wireless Sensor Networks

Eryk Schiller, Ramon Huber, Burkhard Stiller
Communication Systems Group CSG, Department of Informatics IfI, University of Zürich UZH
Binzmühlestrasse 14, CH—8050 Zürich, Switzerland
Emails: [schiller|stiller]@ifi.uzh.ch, ramon.huber@uzh.ch

***Abstract*—While Wireless Sensor Networks (WSN) offer potentials, their limited programmability and energy-limitations determine operational challenges. Thus, a TinyIPFIX-based system was designed, such that this application layer protocol is now usable to exchange data in WSNs efficiently. The system implementation in MicroPython is simple and efficient in comparison to a lower level programming language, while displaying valuable properties in terms of overhead and power efficiency. Furthermore, it demonstrates that MicroPython may pave the way towards Network Function Virtualization (NFV) on Internet-of-Things (IoT) devices by providing highly portable software functions implemented in a high-level programming language.**

***Index Terms*—Wireless Sensor Networks, Internet-of-Things, TinyIPFIX, Espressif ESP32-WROOM-32D, Network Function Virtualization (NFV).**

I. Introduction

Wireless Sensor Networks (WSNs) support many beneficial use cases such as agriculture, energy, health, smart city, smart home, or supply chain monitoring settings [4]. With such potentials provided by WSNs, there also exist major challenges preventing from their massive deployments. One such challenge is *(a)* the limited programmability of sensor devices. Very constrained devices, heavily limited in terms of CPU cycles, RAM, 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.

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

Thus, the Espressif ESP-WROOM-32D [11] module was selected and which offers extended capacity. ESP-WROOM-32D supports high level programming languages, such as MicroPython [2]. A MicroPython implementation allows for a seamless execution of network functions (*e.g.,* TinyIPFIX) on a broad spectrum of Internet-of-Things (IoT) devices paving the way toward Network Function Virtualization (NFV) on IoT nodes. The ESP-WROOM-32D device was enriched with the Digi XBee platform [7], which supports a highly constrained and energy efficient IEEE 802.15.4 network. Furthermore, this work deploys the Tiny Internet Protocol Flow Information Export (TinyIPFIX) [12], which supports a long operation on a single battery charge in comparison to regular application protocols, such as Message Queuing Telemetry Transport (MQTT) or Hypertext Transfer Protocol (HTTP).

This paper's remainder is structured in the following way. Section II discusses technologies used for this approach and background of this work. While Section III discusses different design decisions, Section IV and Section V briefly discuss and evaluate the implementation resp. Finally, Section VI summarizes the work and outlines future work.

II. Background

The Internet Protocol Flow Information Export (IPFIX) [5], [6] is an application layer protocol, with the purpose to send information about traffic flows in a network. IPFIX organizes data into Template and Data records. This way, redundant meta information, which is being sent in Template messages, may be sent less often than the actual data sent in Data messages.

TinyIPFIX [12] is an application layer protocol, being derived from IPFIX and which is optimized for constrained WSNs. TinyIPFIX has less overhead than IPFIX, however, it is also based on the unidirectional push-based communication paradigm. As with IPFIX, TinyIPFIX splits its messages into Template and Data messages. To save energy, data messages are sent more often than template messages, while template messages (meta-data) only contain information on the decoding of corresponding data messages.

III. TinyIPFIX-based System Architecture

Important design decisions do show an impact on the resulting architecture of the new approach.

A. Sensor Network (TinyIPFIX)

The TinyIPFIX network distinguishes three types of devices, *i.e.,* End Devices, Concentrators, and Collectors. Transmissions are always directed from an End Device to a Concentrator or from a Concentrator to the Collector, but never the other way around (from a Collector to a Concentrator or from a Concentrator to an End Device). Packets created on End Devices are sent either toward a Concentrator or directly to the Collector. To send, an End Device, *i.e.,* ESP32 devices [11], passes the data packet to the IEEE 802.15.4 device over the Universal Asynchronous Receiver-Transmitter

(UART) connection, which in turn sends it using the IEEE 802.15.4 protocol. A packet sent toward a Concentrator or the Collector first arrives at the corresponding IEEE 802.15.4 adapter and is forwarded over UART to its attached Concentrator or Collector node. IEEE 802.15.4 was selected as the transmission protocol, because it is a widely used standard for indoor environments and has very low power requirements. TinyIPFIX packets arriving at a Concentrator are aggregated with other arriving packets and TinyIPFIX messages derived on the Concentrator using their own sensors. Once the desired number of distinct TinyIPFIX packets was aggregated, the aggregated packet is sent to the Collector. Moreover, it is also possible to install multiple Concentrators on the way 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 a next Concentrator or the Collector.

B. TinyIPFIX Collector

Packets from the Sensor Network arrive at the IEEE 802.15.4 interface of the Collector. The IEEE 802.15.4 interface passes the packet over the Universal Serial Bus (USB) to a non-constrained Device. If a packet containing Template Records arrive, the unknown Template Records is stored, while the known ones are ignored. If a packet contains Data Records, the Data is extracted from the packet using known Templates already stored. The Data is shared with other components using a Publish/Subscribe (Pub/Sub) broker, *e.g.,* Zero Message Queue (ZMQ) [3].

The Collector provides a Pub/Sub network that handles the distribution of the data that arrived from the WSNs. Within the Pub/Sub engine, the data is published using the corresponding TinyIPFIX Set-Id as the Pub/Sub Topic. Applications may then subscribe to Set-Ids of interest and process the data further on. The Pub/Sub engine allows for easy implementations, while an application can access the data desired with just a few lines of code, without needing to worry about the TinyIPFIX Application Programming Interface (API).

IV. Implementation

The proof-of-concept supports two applications: one 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.

A. ESP32 Setup

This work uses two ESP32-based boards, *i.e.,* ESP32 DevKitC V4 [1] and SuperB [10]. The ESP32 DevKitC V4 features a USB port on its own; for the SuperB the Sparkfun XBee Explorer Dongle was used (*i.e.,* an USB-to-XBee layout connector). Both ESP32-based devices are equipped with MicroPython (*i.e.,* appropriate firmware) [2] to be able to parse code written in MicroPython. The code stored in the *main.py* file on the ESP32 device is automatically executed upon every boot.

B. ESP32-XBee Connection

UART is used to connect the ESP32-based device with Digi XBee [7] (the IEEE 802.15.4 communication device). To this end, two lines need to be connected between General Purpose Input/Output (GPIO) pins of ESP32 and RX/TX pins of Digi XBee. Additionally, two lines are used to power up the XBee (power and ground pins) using ESP32's power and ground pins. Should an ESP32 be used as the End Device, two additional GPIO lines need to be used to control the XBee awake and sleep periods, *i.e.,* Sleep-Control input and Clear-To-Send (CTS) output. When the Sleep-Control changes from high to low, the XBee device wakes up, and informs that it is ready to send using the CTS pin.

C. Configuring XBee Devices

The XBee network has one Personal Area Network (PAN) coordinator (*i.e.,* IEEE 802.15.4) managing the network and multiple end devices. The TinyIPFIX Collector device was chosen to be the Coordinator, while other XBee devices became IEEE 802.15.4 End Devices. Many settings can be configured by XBee modules, but particularly interesting here are: Communication Channel (CH), PAN ID (ID), Upper 32 bit of the Destination Address (DH), Lower 32-bits of Destination Address (DL), 16-bit Source Address (MY), Coordinator Enable (CE), MM (MAC Mode), defining whether acknowledgements for the transmission are configured, and DIO7 Configuration (D7) enabling the CTS flow control.

D. TinyIPFIX Protocol Implementation

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 [8] may be accessed.

1) TinyIPFIX Helper Functions Class: This class is a 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 bitstrings (*i.e.,* strings of bits).

2) Tiny IPFIX Message Specific Classes: The remainder of classes represent different parts of the TinyIPFIX message structure. The generic Message Class describes a generic message. It converts a Message object instance to a bytes object sent over any underlying network. Every TinyIPFIX Message may be derived with the help of more specific classes, such as Message Header, Template Records Set, or Data Records Set classes.

3) WSN Device Related Classes: The Device Class offers many functions that are all concerned about creating, sending, and receiving Messages as well as setting the clock or using sensors. The End Device Class (for end devices) and the Concentrator Class (for concentrators) both inherit from the Device Class. For example, the End Device implements a *deep sleep* functionality, which requires some parameters to be stored on flash permanently when the device enters the sleep mode (*i.e.,* is powered off).

4) Collector and Application Related Classes: Three Classes were created and named Collector, Subscriber Database, and Subscriber Print. These three classes are implemented using Python as a programming language, while all WSN-related classes depend on MicroPython. The Collector Class maintains a list of templates storing all templates known to the Collector. Furthermore, it offers the ZMQ port, context, and socket used to publish data. Finally, the two Application classes subscribe to the topic of interest allowing them to receive messages based on their subscription. One application stores all data received in a database, while the other one prints all data received on the console.

V. Evaluation

All devices set the same CH, ID, and DH to values 26, 7,385, and 0, respectively (*cf.* Sect. IV-C). The value of MY and DL is variable on every device. CE is set 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*. 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 indicate whether the XBee is ready to send data. All seven devices form a tree structure with End Devices (*i.e.,* A.1, A.2, B.1, and B.2), Concentrators (*i.e.,* A and B), and Collector being leafs, branches, and the root, respectively configuring MY and DL accordingly. They use simple static tree routing for packet forwarding. Devices A.1 and A.2 (resp. B.1 and B.2) belong to Concentrator A (resp. B) branch. As an example End Device A.1 forwards packets to Concentrator A, which, in turn, forwards its packets to the Collector.

A. Data Overhead

The overhead of a transmission in TinyIPFIX is now compared against the overhead of a simple Type-Length-Value (TLV) approach [9]. In the TLV data representation, a message is constructed by combining the type, length, and value elements as message fields.

Under the assumption that one record consists of a 4 Byte float read-out (*e.g.,* temperature) and a timestamp, which equals 5 Byte that correspond to regular timestamps used by MySQL, a TLV Packet would maintain 3 Byte overhead per value (*i.e.,* 2 Byte for the type field and 1 Byte for the length field) accompanying both: the 4 Byte temperature read-out and the 5 Byte timestamp. In total, 6 Byte of overhead and 9 Byte of data are sent in such a record, which results in 40% overhead per record.

A TinyIPFIX Template message shows a 21 Byte overhead when a TinyIPFIX template message is sent with two field specifiers (*i.e.,* one record). The TinyIPFIX Data message has 3 Byte overhead, a 2 Byte overhead per field, and 9 Bytes data. Therefore, the overhead of TinyIPFIX is dependent on the number of data records per data message sent and the frequency of template messages, *i.e.,* how many data messages are sent per every template message, *cf.* Figure 1(a). Therefore, TinyIPFIX may in specific settings outperform the TLV reporting method.

B. 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 runs for one hour. Multiple measurements were performed, where the distance between devices was set to 2, 5, 10, and 20 m, with a different MM setting (*cf.* Sect. IV-C), which controls the IEEE 802.15.4 acknowledgement (ACK). This work measured the number of data record sets arriving at the Collector and Concentrators as the function of packets originating at End Devices.

All devices in the WSN were configured to create a data record set every 10 s and to send a data message as soon as two data record sets were created. Figure 1(b) shows the results of these measurements. Overall, it can be observed that most of these data record sets sent successfully arrive at the destination. These measurements also confirm that the implementation of TinyIPFIX was successful and reliable providing TinyIPFIX messages towards the Collector.

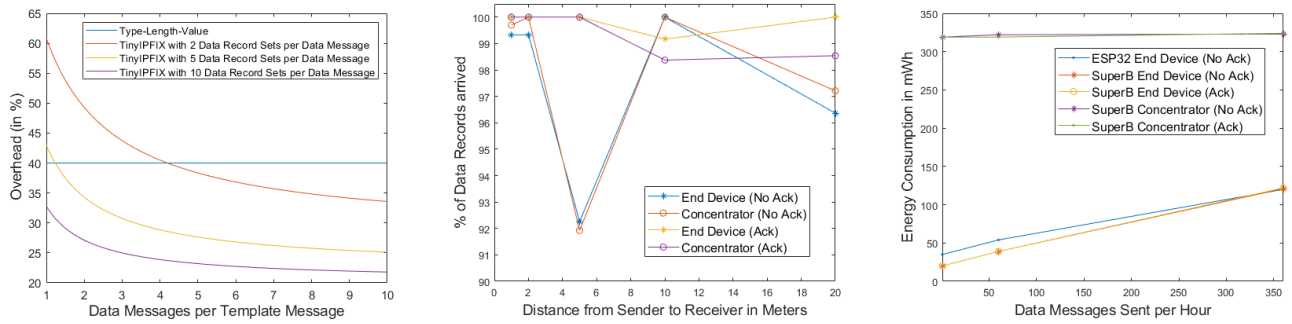
C. Energy consumption

A power meter was used to measure the energy consumption of the devices in different configurations. Table I 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 as well. 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.

TABLE I
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 1(c) 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 loose any arriving messages. The energy consumption mainly depends on the time that devices are in the sleep or deep sleep mode. Using an alkaline battery of 4,200 mWh, the SuperB device reporting one data record per hour could last



(a) TinyIPFIX and TLV Overhead Comparison

(b) Transmission Reliability

(c) Energy Consumption based on the Message Sending Interval

Fig. 1. Performance Characteristics of Python-based TinyIPFIX in WSNs.

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 mW.

VI. Summary and Conclusions

In this work, a TinyIPFIX platform was implemented using Espressif ESP-WROOM-32D devices. TinyIPFIX was selected as a data transport mechanism for the Wireless Sensor Network (WSN). 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 ESP32 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. Finally, all components of the network were implemented with the help of MicroPython (*i.e.,* End Devices, Concentrators) or Python (*i.e.,* the Collector).

In conclusion, in these application scenarios TinyIPFIX maintains a smaller data overhead than the regular TLV data transfer. Furthermore, it has been demonstrated experimentally that the Python-based TinyIPFIX works well in a home-based IEEE 802.15.4 network providing almost a 100% delivery ratio. Moreover, the energy consumption of those devices running TinyIPFIX has been evaluated successfully, since the main 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). Furthermore, the TinyIPFIX module was provided in MicroPython as a software module. It paves the way towards Network Function Virtualization (NFV) on Internet-of-Things (IoT) devices by providing a highly portable protocol implementation.

Acknowledgements

This paper was supported partially by *(a)* the University of Zürich UZH, Switzerland, and *(b)* the European Union's Horizon 2020 Research and Innovation Program under Grant Agreement No. 830927, the CONCORDIA project.

References

[1] "ESP32-DevKitC V4 Getting Started Guide," https://docs.espressif.com/projects/esp-idf/en/latest/esp32/hw-reference/esp32/get-started-devkitc.html, Accessed: 2020-11-07.

[2] "MicroPython Homepage," https://micropython.org/, Accessed: 2020-09-04.

[3] "ZeroMQ Messaging Patterns - Publish/Subscribe," https://learning-0mq-with-pyzmq.readthedocs.io/en/latest/pyzmq/patterns/pubsub.html, accessed: 2020-10-05.

[4] I. F. Akyildiz, W. Su, Y. Sankarasubramaniam, and E. Cayirci, "Wireless Sensor Networks: a Survey," *Computer networks*, vol. 38, no. 4, pp. 393–422, 2002.

[5] B. Claise and B. Trammell, "Information Model for IP Flow Information Export (IPFIX)," Internet Requests for Comments, RFC Editor, RFC 7012, 09 2013. [Online]. Available: https://www.rfc-editor.org/rfc/rfc7012.txt

[6] B. Claise, B. Trammell, and P. Aitken, "Specification of the IP Flow Information Export (IPFIX) Protocol for the Exchange of Flow Information," Internet Requests for Comments, RFC Editor, RFC 7011, 09 2013. [Online]. Available: https://www.rfc-editor.org/rfc/rfc7011.txt

[7] *XBee/XBee-PRO S2C Zigbee RF Module User Guide*, Digi International, 01 2020, aG, https://www.digi.com/resources/documentation/digidocs/pdfs/90002002.pdf.

[8] R. Huber, "TinyIPFIX for ESP32, GitHub Repository," https://github.com/ramonhuber/TinyIPFIX-for-ESP32, Oct. 2020.

[9] T. Kothmayr, "Data Collection in Wireless Sensor Networks for Autonomic Home Networking," *Bachelor Thesis, Department of Computer Science, Technische University of Munich, Munich, Germany*, 2010.

[10] Macchina LLC, "SuperB ESP32 Breakout - Overview," https://docs.macchina.cc/superb-docs/hardware, Dec. 2017, Revision: R0, Accessed: 2020-09-20.

[11] A. Maier, A. Sharp, and Y. Vagapov, "Comparative Analysis and Practical Implementation of the ESP32 Microcontroller Module for the Internet of Things," in *2017 Internet Technologies and Applications (ITA)*. IEEE, 2017, pp. 143–148.

[12] C. Schmitt, B. Stiller, and B. Trammell, "TinyIPFIX for Smart Meters in Constrained Networks," Internet Requests for Comments, RFC Editor, RFC 8272, 11 2017. [Online]. Available: https://www.rfc-editor.org/rfc/rfc8272.txt