Python-Based TinyIPFIX in Wireless Sensor Networks

Ramon Huber, Eryk Schiller, and Burkhard Stiller

Communication Systems Group CSG
Department of Informatics IfI, University of Zürich UZH
Binzmühlestrasse 14, CH—8050 Zürich, Switzerland
`ramon.huber@uzh.ch, [schiller|stiller]@ifi.uzh.ch`

Abstract. This work designs and implements a new TinyIPFIX application layer protocol in Micropython, which enables low power communication in a WSN. The system is deployed on the novel Espressif ESP32-WROOM-32D Internet-of-Things (IoT) platform. The protocol is implemented and evaluated in an indoor smart-home scenario. It displays promising performance and reasonable overhead. Furthermore, it demonstrates that Micropython may pave the way towards Network Function Virtualization (NFV) on IoT devices by providing highly portable software functions implemented in a high-level programming language.

Keywords: Wireless Sensor Networks · Internet-of-Things · TinyIPFIX · Espressif ESP32-WROOM-32D · ESP32 · Micropython · Network Function Virtualization (NFV)

1 Introduction

The deployment of constrained Internet-of-Things (IoT) devices became ubiquitous due to their low price. Such devices perform the role of sensor modules, which are designed to sense the environment and send the information to other system components (*e.g.,* applications residing in the cloud). Sensors organized in complex structures, *e.g.,* multi-hop networks, are referred to as Wireless Sensor Networks (WSN). WSNs may support many different areas, *e.g.,* agriculture, energy, health, industrial, smart city, smart home, or supply chain [4]. As an example, the combination of smart home/city in energy applications may lead to reduced energy wastes by measuring the energy needs of each household and planning the energy production accordingly. However, WSNs often bring challenges. Two main challenges recognized in this work are *(a)* a limited programmability of sensor devices, while the implementation of advanced features requires tedious low level programming and *(b)* the energy efficient operation, while most of the sensors are battery powered.

Three approaches were chosen to cope with these challenges: *(1)* a new generation of sensor devices relaxes constraints on CPU, RAM, and networking, *(2)* a high-level programming language efficiently implements energy efficient operations, and *(3)* an optimal protocol sends data within the WSN, keeping overhead at a minimal level and, therefore, saving energy.

The Espressif ESP-WROOM-32D (*i.e.,* ESP32) [8, 11] module was selected, since it is programmed with the help of high-level languages, such as Micropython [1] or JavaScript. Additionally, the Digi XBee platform [7] was connected to the ESP32 module to add the IEEE 802.15.4 network support [3]. Note that IEEE 802.15.4 is energy efficient and widely used for indoor environments. Furthermore, the Tiny Internet Protocol Flow Information Export (TinyIPFIX) [12] was implemented on this platform. This protocol sends metadata and actual sensor data in separate messages and reduces overhead in comparison to other application layer protocols such as Message Queuing Telemetry Transport (MQTT) or Hypertext Transfer Protocol (HTTP) frequently used in IoT applications.

This paper's remainder is structured as follows. Section 2 summarizes technologies used in this work. While Section 3 discusses different design decisions and outlines the implementation, Section 4 preliminarily evaluates it. Finally, Section 5 summarizes the paper and outlines future work.

2 Background

IPFIX [5, 6] is a protocol originally designed to send information about traffic flows in the network. However, its properties suit WSNs, too, since it uses two messages, *i.e.,* Data Messages and Template Messages. Template Messages contain meta-information, while Data Messages handle the actual information. Furthermore, IPFIX is push-based, thus, a sender sends data when available, however, the receiver is unable to request data from a sender.

TinyIPFIX [12] is an application layer protocol derived from IPFIX. It is push-based and distinguishes Template and Data Messages. Template Messages contain meta-information for the decoding of Data messages as well. While Template Messages rarely change, they are sent less often than Data Messages. A Data Message points towards a corresponding Template Message, which contains the information on how to decode a given Data Message. However, Data Messages maintain a limited amount of meta-information. TinyIPFIX saves energy by reducing overhead using the Template and Data Messages. Furthermore, TinyIPFIX one-way communication paradigm allows the nodes to go sleep, thus reducing the energy consumption on IoT devices. Thus, TinyIPFIX is well suited for WSNs, in which it supports an energy efficient operation.

3 TinyIPFIX-based System Architecture

This architecture consists of a network, a collector, and applications.

3.1 Sensor Network (TinyIPFIX)

TinyIPFIX End Devices measure the environment using a sensor and create a TinyIPFIX Data Message containing the data sensed. The Data Message is sent to a Concentrator or directly to the Collector in the Network. To send data, the

ESP32 device [8, 11] passes the message to the IEEE 802.15.4 [3] device using a Universal Asynchronous Receiver-Transmitter (UART) connection. The IEEE 802.15.4 device sends it toward the desired destination using the IEEE 802.15.4 Physical Layer (PHY) protocol. At the destination, the packet is again received by the IEEE 802.15.4 device and forwarded over the UART.

TinyIPFIX Concentrators may sense the environment using a local sensor, but additionally they can receive messages from remote nodes in the network and aggregate multiple TinyIPFIX messages together into one message. This results in fewer larger messages sent in the network, which saves energy. Ultimately, all messages need to arrive at the network Collector equipped with the IEEE 802.15.4 interface. Each sensing device may send data directly to the Collector or over one or multiple Concentrators in a multi-hop manner.

3.2 TinyIPFIX Collector

All messages from the WSN arrive at the Collector, *i.e.,* the unconstrained device running the Collector module. The Collector and the overlying Publish Subscribe (Pub/Sub) Network are responsible for providing the data received further on. When a Template Message arrives at the Collector, the template is stored only, if a new template has arrived, otherwise the template received is ignored. When a Data Message is received, it is decoded using a known template. The data received is published using a Pub/Sub message broker, *i.e.,* Zero Message Queue (ZMQ) [2]. ZMQ uses the TinyIPFIX Set-Id as the Topic. Applications can subscribe to these topics of interest and process data without a need to implement any TinyIPFIX infrastructure.

3.3 Applications

Two applications were developed as a proof-of-concept. One prints the data received from the WSN to the console on the application server, the other one stores the data in a Relational Database Management System (RDBMS) Database. All details of the system components implemented are accessible online [9].

4 Evaluation

Four End Devices, two Concentrators, and one Collector were placed in a home environment in order to conduct a realistic evaluation of the system.

4.1 Network Configuration

Every pair of End Devices sends their messages toward a distinct Concentrator. These two Concentrators aggregate messages from these End Devices together with their own messages and send resulting messages to the Collector.

4.2 Data Overhead

For the data overhead evaluation, TinyIPFIX is compared to the simple Type-Length-Value (TLV) approach [10]. The Type denotes the type of value that is being sent, *e.g.,* a temperature reading, the length denotes the length of the value field, while the value field carries the actual reading. A TLV message is built by combining one or several TLV components. Should two sensor readings be sent, the message concatenates two TLV entries all together.

TinyIPFIX supports up to 2^{16} Internet Assigned Numbers Authority (IANA) predefined information elements. Additionally, TinyIPFIX can distinguish 2^{32} enterprises, in which 2^{16} types per enterprise may be defined. Assuming that only one enterprise number is used in the network, TinyIPFIX supports 2^{16} different value types. To provide a fair comparison between TinyIPFIX and the TLV paradigm, this work assumes a 2-Byte long type field. The length field, in turn, is assumed to be 1-Byte long, which equals the regular size of the length field in TinyIPFIX. A TinyIPFIX Template Message contains 7 Byte of fixed overhead and 7 Byte of overhead for every field specifier. A TinyIPFIX Data Message contains 3 Byte of fixed overhead, 2 Byte overhead for each field specifier, and a variable amount of data. Taking the assumption that a message consists of a 4-Byte float sensor reading (*e.g.,* temperature reading) and a 5-Byte timestamp (*e.g.,* as exactly for MySQL), a message in the TLV format would consist of 6 Byte overhead and 9 Byte of data, which results in a 40% overhead. Under this assumption, a TinyIPFIX Template Message contains a 21 Byte overhead, while a data message contains 3 Byte of overhead and 9 Byte of data.

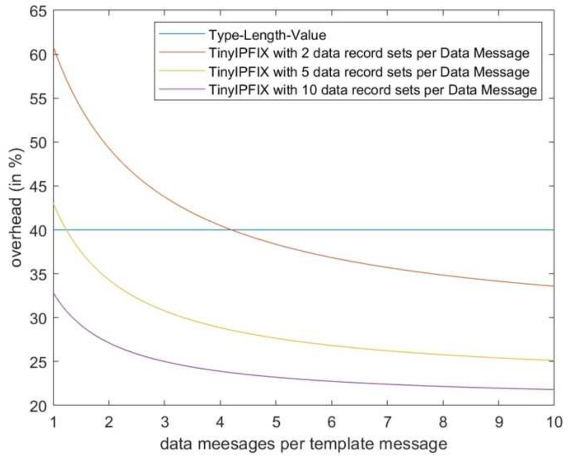

Fig. 1. TinyIPFIX and TLV Overhead Comparison

Thus, the TinyIPFIX overhead (*cf.* Figure 1) depends on how many data items are being sent per Data Message and how many Data Messages are sent per Template Message. Figure 1 visualizes the overhead compared to the TLV paradigm, which displays that TinyIPFIX can significantly decrease the overhead. Especially in a WSN, where templates do not change often (*i.e.,* while templates do not need to be re-sent frequently) and where several data records can be packed into a single data message, TinyIPFIX shows excellent properties.

4.3 Transmission Reliability

To evaluate the transmission reliability, several measurements were performed, too. End Devices were spaced between 1 and 20 m apart. In each distinct measurement, the system ran for 1 h sending a data message every 20 s. The respective measurements showed that packets arrived with a reliability of around 99% (90% in the worst case), demonstrating an excellent real-world prototype.

4.4 Energy Consumption

A power meter was used to measure the energy consumption of End Devices in different configurations. The power meter plugs into a USB port (*e.g.,* a PC), which powers the meter. The device measured 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 milli Watt (mW) and over which period of time. For example, the power consumption of the ESP32 End Device/Concentrator, respectively, resulted for the "Deep Sleep" mode in 35 mW/-, for "Idle" in 190 mW/328 mW, for "Sending" in 344 mW/334 mW, and for "Receiving" in -/390 mW. The implementation of TinyIPFIX in these settings is considered successful, however, the configured energy expenses of ESP32 devices are still high due to the elevated deep sleep power consumption.

5 Summary and Conclusions

A Wireless Sensor Network (WSN) platform was implemented using Espressif ESP-WROOM-32D devices. While two different ESP-WROOM-32D device families were considered, *i.e.,* the Espressif ESP32 DevKitC V4 and the Machhina SuperB, every device is equipped with a Digi XBee board in order to support IEEE 802.15.4 connectivity, which enables low power communication between devices in the WSN. The implementation was performed in high-level languages. MicroPython was used for the implementation of End Devices and Concentrators, while Python was used for the Collector. The TinyIPFIX protocol was implemented as a transport mechanism delivering data from End Devices toward the Collector. The data is distributed to applications from this Collector using a Publish/Subscribe (Pub/Sub) engine implemented with the help of the ZMQ message broker. Two proof-of-concept applications were developed, *i.e.,* receiving and processing messages arriving from the message broker.

This new approach using a high-level language (*i.e.,* MicroPython) for the protocol implementation enabled a faster value creation, which is impossible on older platforms relying on low-level programming languages. Furthermore, TinyIPFIX was implemented in a portable high-level language allowing for high code portability. This Micropython implementation allows for a seamless execution of network functions (*e.g.,* TinyIPFIX) on a broad spectrum of IoT devices paving the way toward Network Function Virtualization (NFV) on IoT nodes.

The system was experimentally verified in a home environment. It showed almost a 100% delivery rate in such a WSN. Furthermore, it was shown that the TinyIPIFX implementation developed features lower data overhead than the Type-Length-Value (TLV) approach.

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. MicroPython Homepage. https://micropython.org, accessed: 2020-09-04
2. ZeroMQ Messaging Patterns - Publish/Subscribe. https://learning-0mq-with-pyzmq.readthedocs.io/en/latest/pyzmq/patterns/pubsub.html, accessed: 2020-10-05
3. IEEE Standard for Low-Rate Wireless Networks. IEEE Std 802.15.4-2020 (Revision of IEEE Std 802.15.4-2015) pp. 1–800 (2020). https://doi.org/10.1109/IEEESTD.2020.9144691
4. Akyildiz, I.F., Su, W., Sankarasubramaniam, Y., Cayirci, E.: Wireless Sensor Networks: a Survey. Computer networks **38**(4), 393–422 (2002)
5. Claise, B., Trammell, B.: Information Model for IP Flow Information Export (IPFIX). RFC 7012, RFC Editor (09 2013), https://www.rfc-editor.org/rfc/rfc7012.txt
6. Claise, B., Trammell, B., Aitken, P.: Specification of the IP Flow Information Export (IPFIX) Protocol for the Exchange of Flow Information. RFC 7011, RFC Editor (09 2013), https://www.rfc-editor.org/rfc/rfc7011.txt
7. Digi International: XBee/XBee-PRO S2C Zigbee RF Module User Guide (01 2020), https://www.digi.com/resources/documentation/digidocs/pdfs/90002002.pdf
8. Espressif Systems: ESP32-WROOM-32D & ESP32-WROOM-32U Datasheet (09 2019), V1.9, https://www.espressif.com/sites/default/files/documentation/esp32-wroom-32d_esp32-wroom-32u_datasheet_en.pdf, Accessed: 2001-05-30
9. Huber, R.: TinyIPFIX for ESP32, GitHub Repository. https://github.com/ramonhuber/TinyIPFIX-for-ESP32 (Oct 2020)
10. 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)
11. Maier, A., Sharp, A., Vagapov, Y.: Comparative Analysis and Practical Implementation of the ESP32 Microcontroller Module for the Internet of Things. In: 2017 Internet Technologies and Applications (ITA). pp. 143–148. IEEE (2017)
12. Schmitt, C., Stiller, B., Trammell, B.: TinyIPFIX for Smart Meters in Constrained Networks. RFC 8272, RFC Editor (11 2017), https://www.rfc-editor.org/rfc/rfc8272.txt