JITTERS IN OPERATING SYSTEMS FOR THE INTERNET OF THINGS

A Thesis Presented to

the Faculty of the Department of Computer Science

University of Houston

In Partial Fulfillment

of the Requirements for the Degree

Master of Science

By

Navneet Pandey

December 2018

JITTERS IN OPERATING SYSTEMS FOR THE INTERNET OF THINGS

Navneet Pandey

Acknowledgements

I would like to thank Dr. Omprakash Gnawali for continuous support in my research. I would also like to thank Hessam Mohammadmoradi, and my lab mates at the Networked Systems Laboratory for providing all the help whenever I needed.

JITTERS IN OPERATING SYSTEMS FOR THE INTERNET OF THINGS

An Abstract of a Thesis

Presented to

the Faculty of the Department of Computer Science

University of Houston

In Partial Fulfillment

of the Requirements for the Degree

Master of Science

By

Navneet Pandey

December 2018

Abstract

The Internet of Things (IoT) is an extension of the internet into the physical world through the use of sensing, actuation, control, and interaction with embedded devices. A large number of IoT devices are being deployed in the world. The emerging applications involving IoT require reliable network connectivity. Latency is one of the most critical network performance metric that will determine the user experience with IoT applications. There are two aspects of latency metric – the overall delay and the jitters. Most of the focus is on low delay but many applications, especially the ones with real-time-like requirements, also need low jitters in latency to have predictable protocols or interactions at the system level.

This thesis presents a study of jitters in the IoT operating systems observed through various networking-related operations and systems. The execution and performance of the application can be greatly affected by the characteristics of an Operating System (OS) in the IoT system. This thesis presents a study of network stack performance, layer-wise packet trace, and its analysis. The key focus of the analysis is identifying the presence of jitters in the IoT OS and the contributing factors behind their presence. The approach taken by this thesis is that it performs a series of measurement studies of basic applications on IoT hardware and OS platforms. We evaluate this study with two OS – RIOT and Contiki OS and two IoT hardware platforms – IoTLAB-M3 open node and TelosB. This thesis provides guidance on the achievable network performance and characteristics for different system requirements of IoT applications.

Contents

Chapter 1: INTRODUCTION 1

Chapter 2: RELATED WORK 5

Chapter 3: DELAYS AND JITTERS IN IOT 9

Chapter 4: EVALUATION 15

4.1. Hardware Platforms 15

4.2. Operating Systems 16

4.3 Testbed 19

4.4. Experiment Setup 20

4.5. Experiment Design 21

4.6. Experiment Summary 24

4.7. Results 27

Chapter 5: CONCLUSION 50

Chapter 6: REFERENCES 52

List of Figures

Figure 1. Decomposition of the message delivery delays over a wireless network [21].. 11

Figure 2. Processing of Message Reception Event in RIOT OS [39]. 12

Figure 3. Packet Forwarding Process to the Hardware in RIOT OS [40]........................ 12

Figure 4. Packet Forwarding Process from the Hardware in RIOT OS [40]. 13

Figure 5. RIOT Networking Subsystem [44]. 17

Figure 6. Communication Stack in Contiki OS. 18

Figure 7. Layout of IoTLAB-M3 node during an experiment. 22

Figure 8: Timing and Flow of Various Components in an RTT Measurement Experiment. 22

Figure 9. RTT Measured at the Application Layer (IoTLAB-M3, RIOT). 31

Figure 10. RTT Measured at the MAC Layer (IoTLAB-M3, RIOT). 31

Figure 11. RTT Measured at the Application Layer (TelosB, RIOT). 32

Figure 12. Packet Delays between Networking Modules at the Responder. 33

Figure 13. Packet Delays between Networking Modules at the Initiator Node................ 34

Figure 14. Packet Formatting Delays in Different Networking Modules at the Initiator Node. 34

Figure 15. Duration of Thread Runtime at the Initiator Node. 35

Figure 16. Delays in RTT Due to Driver's Send Submodules. 36

Figure 17. Delays Between MAC Layer and ISR during Packet Transmission. 38

Figure 18. RTT at the Driver Level after Initiator Completed the Transmission. 39

Figure 19. Packet Transmission Time at the Responder Node After Packet Reception Until Interrupt Invocation. 39

Figure 20. Delays in Submodules of Driver's “send” Module (Initiator and Responder). 41

Figure 21. Processing Delay at the Responder Node. 42

Figure 22. RTT Measured at the Application Layer. (IoTLAB-M3, Contiki). 44

Figure 23. Comparisons of Deviation Produced by RIOT’s Clock and IoTLAB Timestamps (in µs, Deviation from 0.5 seconds). 46

Figure 24. Packet Reception Time for 20 Packets (in Seconds) (IoTLAB). 47

Figure 25. Reception Times for 20 Packets (xtimer). 48

Figure 26. Round-Trip-Time (calculate with IoTLAB). 49

Figure 27. Round-Trip-Time (calculate with xtimer). 49

List of Tables

Table 1. Summary of RTT Measurement Experiments 24

Table 2. Summary of Other Experiments 26

Table 3. Result Summary 27

Table 4. Delays Between Driver Module and Interrupt Service Routine (ISR) 37

Table 5. Delays in between RIOT Modules (for TelosB) 42

Table 6. Network Stack Traversal Analysis 44

Table 7. Driver Radio Module Analysis 45

Chapter 1

INTRODUCTION

As the Internet of Things (IoT) network grows, the need for reliable communication between sensors also increases. The increasing IoT network has and will result in a variety of different applications and services being used. It would not be wrong to state that some aspect of this huge and growing network would require reliable and deterministic network communication. To ensure this, there is a need for studying and analyzing network measurements. In fact, Quality of Service (QoS) in the field of networking is a frequent topic of research. QoS is a way to measure the performance of a network. QoS is a collective term for a bunch of measurements that include delay, delay variation, bandwidth, and packet loss etc. Delay variation, also known as jitters, is a big factor when it comes to uncertainties in a network.

Jitter is not a new problem, but it is an important area of research in the field of networking. Jitter leads to uncertainty, which in turn leads to non-deterministic outcomes and, ultimately, unreliable communication. Network protocols and applications use timers to determine when they expect to receive a message from the network. For example, retransmission timers in network protocols, or network wait times in application logic. A good understanding of the jitters on the platforms and network would allow an accurate setup of such timers. In case of real time systems or near-real time systems, assessment of variations in latencies is a must, so that they can be compensated to ensure that the system is real time or almost real-time. In case of non-real time systems, jitter analysis would help developers and researchers keep their systems in check and also boost the reliability aspect of networking. Finally, minimizing jitters in IoT systems would not only enhance user experience but would also facilitate the migration of users to the IoT world.

Evaluating QoS would be easy if networks were static. But networks are dynamic, and they keep on changing with time. This makes maintenance difficult. Hence, even unrelated and unconnected changes might affect the QoS measurements. For example, jitter in end-to-end delay or round-trip-time delay might be caused by hardware changes or software changes. Hence, there is this constant need to evaluate networks and keep unwanted delays in check.

QoS evaluation has been a constant topic of research in the past and will be a topic of research in the future as well due to the volatile nature of networks, especially in the case of IoT. Researchers are constantly trying to either eliminate or circumvent jitters to achieve high throughput and real time communications. Numerous studies (RELATED WORK, Page 5) have been performed on delay and jitter analysis. But, none of those studies evaluated jitters in the IoT environment which this thesis does. This thesis uses real data acquired from experiments on real hardware platforms for the analysis. To the best of our knowledge, there is no existing work which evaluates jitter in the IoT environment.

Operating systems and testbeds form the core component of this thesis. An IoT OS is an operating system suited for devices which are restricted in terms of memory, size, power, processing capacity etc. [1]. This thesis analyses the IoT operating systems on multiple fronts to pinpoint the cause of jitters, if any. This work contains experiments such as packet trace via extensive logging, determining effects of interreference, effects of queue sizes, etc. Due to the time constraints, this thesis keeps its focus on RIOT OS [2] and Contiki OS [3] only. The hardware platforms used to perform experiments were IoTLAB-M3 [4] and TelosB [5]. Time Synchronization between nodes would have allowed us to measure and analyze one way end-to-end latencies across the nodes, but since it was beyond the scope of this thesis, we had to use the serial logging script from IoTLAB [6] and limit ourselves to round-trip-time (RTT) measurements. As we will see later in this work, IoTLAB proved less reliable in terms of timestamp logging when compared with operating system's built-in clock timestamping. Our results with Contiki OS are inconclusive at the moment.

The jitter analysis performed for this thesis shows that, despite the use of bare minimum modules, there was a jitter of about 4 ms in RIOT OS. The analysis revealed that the cause of this jitter was interrupt handling. This theory was supported by the fact that TelosB lacked the full interrupt handling process and that the jitters were considerably less, allowing us to suggest that in RIOT OS, the jitter is caused by Interrupt Service Routine (ISR) handling. This thesis makes the following contributions:

- We analyze RIOT and Contiki operating system and pinpoint the cause of jitters in the respective OS.
- We point out that using ’Serial Aggregator’ [7] does not provide accurate timestamping of data logs from serial ports.

Chapter 2

RELATED WORK

Numerous researches and studies have been presented on the Quality of Service (QoS) aspect of the networks. QoS includes characteristics such as delays, jitters, throughput, bandwidth, transmission rate and packet loss. Different studies address different combination(s) of these characteristics depending upon the scope of their research. Jitter and delay analysis is a frequent subject of research in the field of networking.

2.1. End-to-End Jitter

Nabil et al. [8], highlighted the analysis of end-to-end delay and jitter over LTE Networks. It reached a conclusion that QoS is heavily dependent on network configurations and geographical size. Bolot et al. [9], performed end-to-end delay analysis and measured round-trip-time by sending small UDP packets. Khosrow et al. [10], performed end-to-end analysis and examined the impact of traffic per node and number of nodes on the end-to-end jitter. Chen et al. [11], performed end-to-end delay analysis and measurement on TelosB for TinyOS.

Karam et al. [12], and Zheng et al. [13], analyzed delays and jitters over Voice over IP (VoIP). Rohani et al. [14], and Jeffrey et al. [15], evaluated network performance of VoIP over the WiMAX network. Additionally, the study [15] analyzed the correlation of network performance parameters with distance and carrier-to-interference-noise ratio (CINR)**,** which the authors say can be used for developing reliable semi-empirical models**.** Giorgio et al. [16], measured the end-to-end latency and jitters in a Carrier Ethernet Network exploiting Ethernet OAM tools. Y. Frank et al. [17], presented a simple scheme to measure end-to-end delays and jitters in an ATM (Asynchronous transfer mode) network via cell delays instead of timestamps using Attila Traffic Analyzer. Soumen et al. [18], performed end-to-end jitter and packet analysis in operational network environment. The evaluation was done using a testbed scenario created on an existing automation system.

Xia et al. [19], performed network measurement like throughput, bandwidth, delay jitters and packet loss using active method of injecting test flow data into the network. This study tested TCP/UDP protocols. Cathy et al. [20], studied the time-dynamic behavior of delay jitter.

2.2. Time Synchronization with Network Jitter Analysis

Stefano et al. [21], performed Statistical characterization of packet jitter in real heterogenous networks, and analysis data by means of Modified Allan Variance (MAVAR) and Maximum Time Interval Error (MTIE). Flood Time Synchronization Protocol (FTSP)

[22] described a protocol for resource limited wireless platform applications. FTSP considered uncertainties in radio message delivery to fulfil its objective. Glossy [23] introduced a novel flooding architecture for wireless sensor networks. The architecture exploited radio interference in a constructive way to overcome the disadvantages of rapid flooding. The study also evaluated the architecture using three wireless sensor testbeds. Jeronimo et al. [24], analyzed the impact of periodic, non-interruptible tasks. The study also presented a modified median filter and a mathematical model to deal with time synchronization errors and considered the impact created by packet transmission jitters.

2.3. Real-Time Based Delay Analysis

TSMAC [25], a new protocol, proposed to support real time data transfer over IEE 802.15.4. The protocol is based on timeslots and maintains throughput as its primary priority to realize a smart home environment. Jitterbug [26] a toolbox, presented an option to analyze real time control performance. The tool allows building of different models based on measurement parameters. Chen et al. [27], performed a subjective evaluation of network via OneClick, a framework to measure quality of network experience. This study assessed the quality of audio and video clips to evaluate QoE (Quality of Experience).

2.4. Jitter Models & Measurement using Protocol Analyzers

Angrisani et al. [28] and [29], measured type A uncertainties in jitter measurements and performed packet jitter analysis. Both studies were aimed at reliable QoS measurements, crucial to evaluate communication networks and used protocol analyzers like SPAPA and GPAPA. Quanxin et al. [30], focused mainly on the jitter feedback mechanism in RTCP and reduced the shortcomings of jitter values in RTCP packets in RTP. Premaratne et al. [31], developed and experimentally verified a jitter model for wired and wireless networks. Rizo et al. [32], described jitters in IP networks via alpha-stable jitter model, estimating jitter-QoS as a function of the number of nodes in the path, the stability index and the jitter dispersion.

2.5. Delay Analysis in IoT/Sensors

Ferrari et al. [33], evaluated the latencies in communication between cloud and nodes in Industrial IoT environment. Nacer et al. [34], presented a study on wireless integration of Wireless Sensors Networks for Internet of Things and evaluated presence of jitter with network traffic as the parameter. IoT-Lite [35], a lightweight semantic model for Internet of Things, proposed rules for scalable semantic models and assessed IoT-Lite in terms of round-trip-time delays. Kelly et al. [36], proposed an implementation of IoT for monitoring environmental condition in homes. The study also evaluated the implementation in terms of reliability and throughput. Leapfrog Collaboration [37] introduced a communication mechanism that takes advantage of promiscuous overhearing via parallel transmission to improve end-to-end reliability and allow deterministic deliveries. The study also performed a delay and jitter evaluation using Contiki-OS. Contiki [38], an operating system for tiny networked sensors was introduced. The OS was evaluated on multiple parameters, including pre-emption, OTA transmissions and round-trip delays.

Chapter 3

DELAYS AND JITTERS IN IOT

Radio message end-to-end delays and round-trip-time delays cannot be considered completely deterministic. “End-to-end delay” refers to the time taken by a packet to transmit from the application layer of the sender to the application layer of the receiver, whereas “round-trip-time” is the time taken by a packet to travel from sender to receiver and back to the sender’s application layer. Both end-to-end latency and round-trip-time have some non-deterministic components, producing variance in delays commonly known as network jitter. Two major aspect of jitter analysis by this thesis were radio component-based delays and operating system-based processing delays. Most of the contents in this section have been borrowed from previous works, [22] and [11]. But this section presents all the components responsible for jitters within a network in one place.

3.1. Delays due to wireless components

3.1.1. Access Time

This represents the waiting period between the time when frame is ready for transmission and the time when the actual transmission begins. This is one of the non-deterministic components in the wireless network.

3.1.2. Encoding/Decoding Delay

These represent the time it takes for the radio chip to convert the data from electromagnetic waves to binary form and vice versa. This delay is mostly deterministic.

3.1.3. Transmission Delay

This component represents the time it takes for the sender to transmit the message. This delay is dependent on the payload and the radio itself.

3.1.4. Propagation Delay

This represents the actual propagation time (air travel time). This is the most deterministic component of a wireless network.

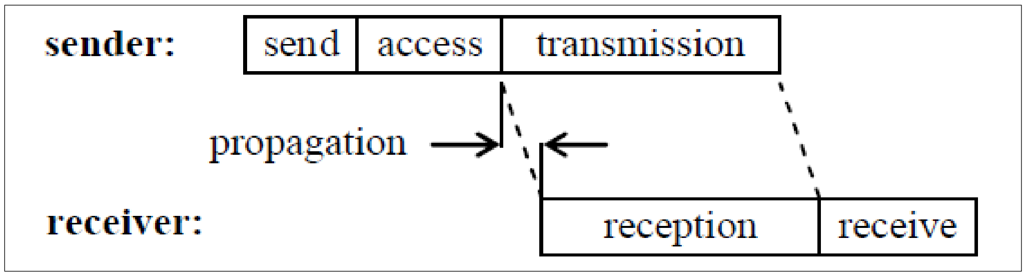

Figure 1. Decomposition of the message delivery delays over a wireless network [21].

3.2. Delays in Operating System

3.2.1. Interrupt Handling Time

This represents the delay between the radio chip's invoking the Interrupt and the microcontroller/CPU's responding to an interrupt. This can be one of the non-deterministic components in the OS.

The following figure describes the process for receiving a packet. The steps followed are: Packet arrives at the device after strobe packets inform the receiver about incoming transmission. Driver handles the generated interrupt and informs the CPU about completion of the reception process, after which the packet goes through all the layers of the networks stack until it reaches the top-most layer, i.e., the application layer.

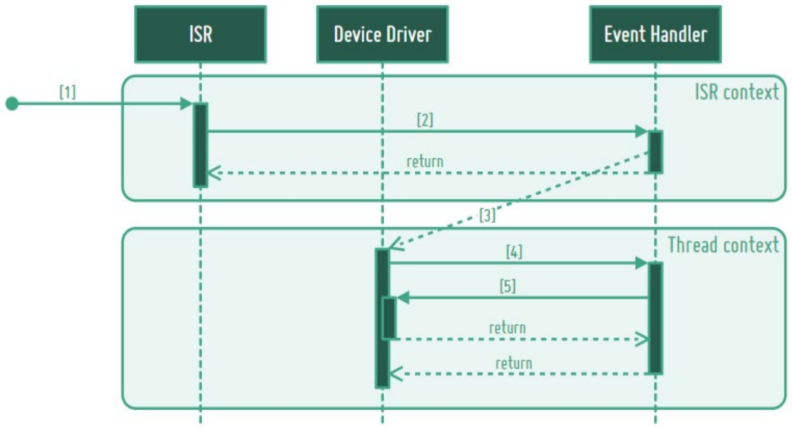

Figure 2. Processing of Message Reception Event in RIOT OS [39].

3.2.2. Send Time

This is a period of time which represents the time taken by the OS to prepare, process, and propagate the packet within the OS. It can be considered as the delay between application layer and radio. This was a key aspect of jitter analysis performed by this thesis. The figure below describes the process of packet flow from the MAC layer to the hardware.

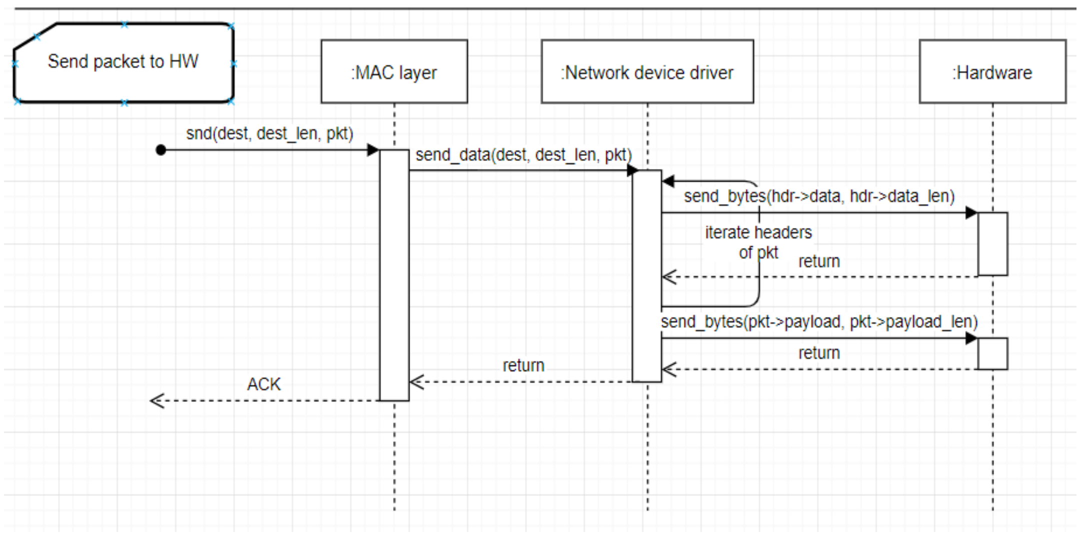

Figure 3. Packet Forwarding Process to the Hardware in RIOT OS [40].

3.2.3. Receive Time

This period represents the time taken by the OS to propagate and process the packet within the OS until the packet reaches the application layer. It can be considered as the delay between the radio and the application layer. This was another key aspect of jitter analysis performed by this thesis. The figure below describes the process of packet flow from the hardware to the MAC layer.

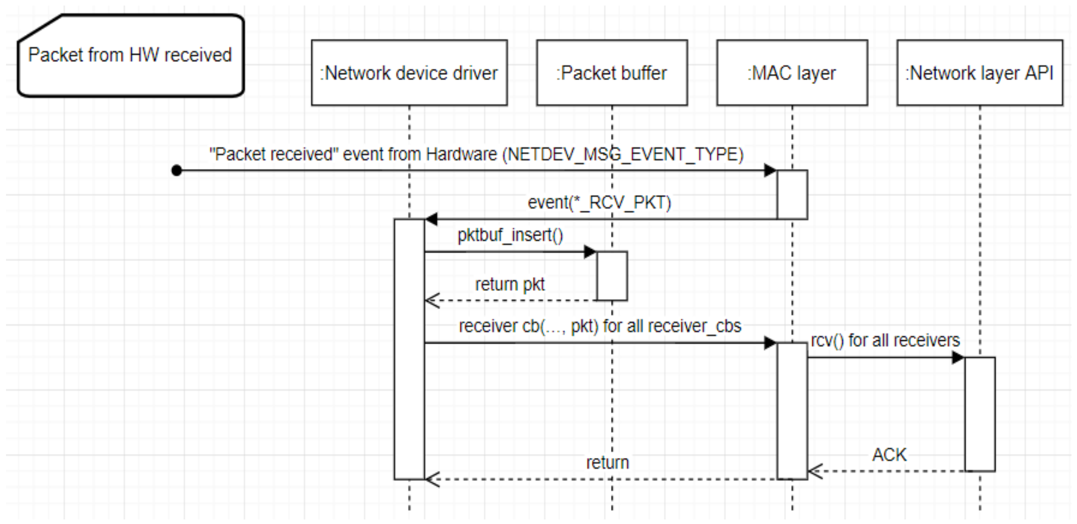

Figure 4. Packet Forwarding Process from the Hardware in RIOT OS [40].

3.2.4. Queuing delay

Queueing delay occurs when transmission doesn't happen fast enough, and packets are queued in the OS. During the evaluation process, we took this aspect out of consideration by keeping the data rate low, thus, providing enough time for the OS to process the previous packet(s).

3.2.5. Network Stacks

A network stack can be defined as a layered set of protocols. This is an important component of an operating system because the characteristics of a network stack govern the communication process of the operating system. Usually, OS in the IoT environment offers default network stack (RIME in Contiki, GNRC in RIOT, etc.). But they also allow the use of custom network stacks such as OpenThread, emb6, IwIP, uIP and many more. Depending upon the choice the network stack, latency and jitter could be different.

Chapter 4

EVALUATION

This section first presents a discussion of the Operating Systems and the IoT hardware platforms used for the evaluation of jitters. Then, we discuss the IoTLAB Testbed before describing the experimental setup and results.

4.1. Hardware Platforms

4.1.1. IoTLAB-M3

The M3 open node is based on a STM32 (ARM Cortex M3) micro-controller. M3 open node consists of a 32-bit processor, an ATMEL radio interface (2.4 Hz AT86RF231) and sensors such as ambient light sensor, atmospheric pressure sensor, gyrometer, and magnetometer. It is powered by a 3.7V, 650mAh LIPo battery, boosts a RAM of 64KB, and has an external flash memory of 128 Mbits [42].

4.1.2 TelosB

Developed and published by UC Berkley, TelosB Mote TPR2420 is an open-source platform. TelosB offers features such as USB programming capability, an IEEE 802.15.4-compatible CC2420 radio with integrated antenna, a low-power MCU with extended memory and an optional sensor suite. TelosB boasts an 8 MHz TI MSP430 microcontroller with 10kB RAM and 1MB external flash memory [5] [43].

4.2. Operating Systems

4.2.1. RIOT OS

RIOT [2] is a free, open-source operating system developed by grassroots community companies and hobbyists. Most of the content in this section has been borrowed from the work [44]. Key features offered by RIOT:

(i) A modular architecture with generic interfaces for plugging in drivers, protocols, or entire stack.

(ii) Support for multiple heterogeneous interfaces and stacks that can operate concurrently.

(iii) GNRC, its cleanly layered, recursively composed default network stack (Figure 1).

RIOT [44] has 2 API interfaces, "Sock" and "Netdev", for interactions external to the GNRC network stack, whereas it offers an API called "netapi" for communication within the stack. "Netdev" API allows RIOT to abstract the driver level design aspect. Sock is an application-level, user-friendly API that allows external communication with external applications and libraries. “Netapi” enables RIOT to pass messages (MSG_TYPE_SND and MSG_TYPE_RCV) between the GNRC stack layers. RIOT supports third-party network stacks as well, including IwIP, emb6, OpenWSN etc., but this thesis limits the testing scope to GNRC stack only.

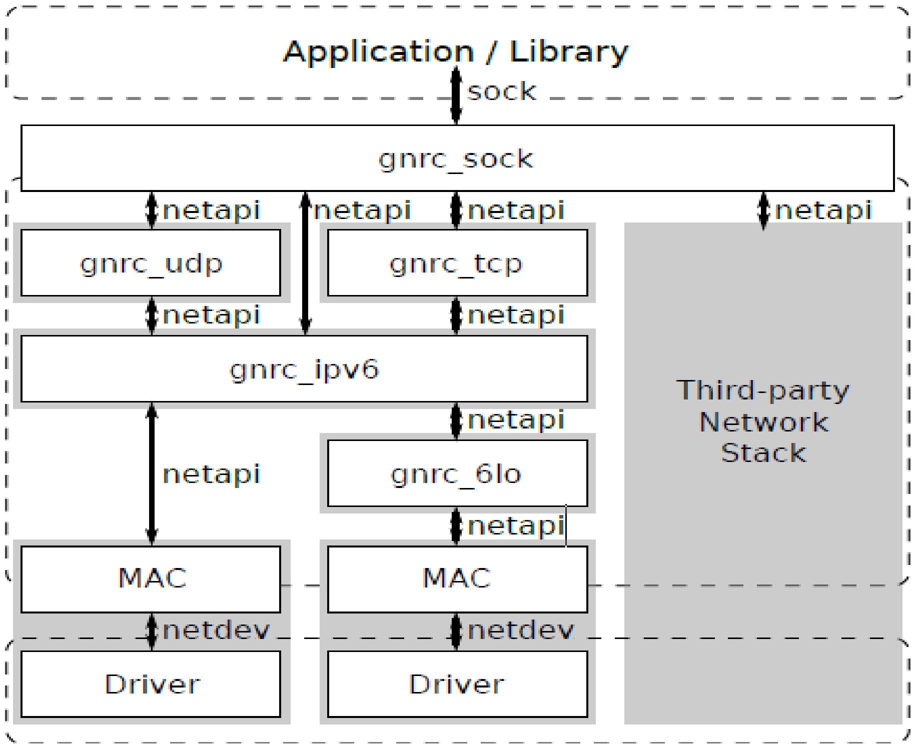

Figure 5. RIOT Networking Subsystem [44].

4.2.2. Contiki OS

Contiki [3] is an open-source operating system for the Internet of Things. Contiki is a powerful toolbox for building complex wireless systems and connecting tiny, low-cost, low-power microcontrollers to the Internet. Contiki is built around event-driven Kernel and is implemented in the language C. Dynamic loading, pre-emptive multithreading and portability are some of the other key features of Contiki [38].

In Contiki, communication is implemented as a service in order to enable run-time replacement. Implementing communication as a service also provides for multiple communication stacks to be loaded simultaneously. Contiki improved upon the uIP network. In addition to uIP, Contiki supports the RIME network stack. Most of the content has been sourced from the Contiki website and a work on Contiki. [3] [38].

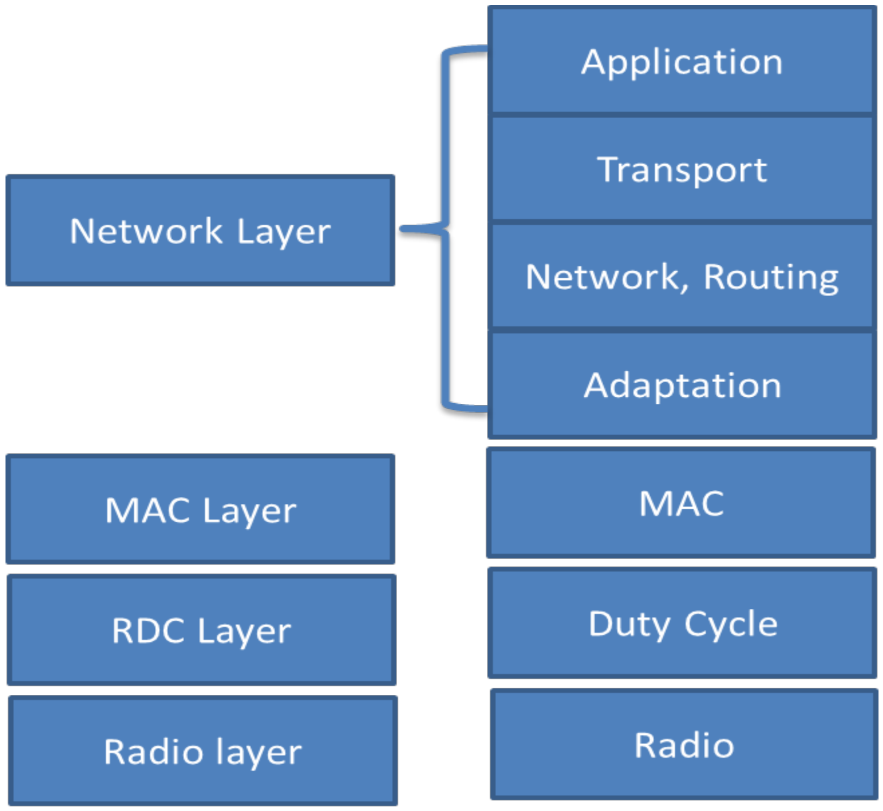

Figure 6. Communication Stack in Contiki OS.

4.3 Testbed

We performed experiments related to IoTLAB-M3 nodes on the IoT-LAB Testbed.

4.3.1. IoTLAB Testbed

It is an open testbed, a part of the FIT (Future Internet of Things) platform. The IoT-LAB infrastructure consists of testbeds located at six different sites across France. The 1791 nodes available for the users include hardware nodes M3, A8, and WSN430. Some of these nodes are mobile as well [41].

4.3.2. Serial Aggregator

The "Serial Aggregator" is a python script, part of the IoTLAB aggregation tool set, which allows simultaneous aggregation of data results from multiple nodes. It connects to several TCP connections and handles the received data by printing the data to "stdout" with the current timestamp.

Serial Aggregator [7] is a widely used logging module used by everyone who uses the IoTLAB testbed to perform experiments; this is the only way to capture the data from serial nodes. We noticed an issue in using the timestamps attached with the data when the logs were dumped by the Serial Aggregator. It turns out that the timestamps need to be used with care, because they introduce additional delay. We performed some experiments to confirm this observation.

4.4. Experiment Setup

Running both RIOT and Contiki operating systems on the Windows platform required the use of virtual machines. Both VirtualBox and VMWare were used to facilitate the virtual machine setup. VMWare was used to compile, and flash builds for TelosB. VirtualBox was required to access the operating systems via SSH (Secure Shell) using Vagrant [45] [46].

An SSH setup was also required from the Windows (host) machine to the IoTLAB testbed to start the logging process [47]. PSCP (PuTTY Secure Copy) protocol was used to transfer the log files from the remote testbed to the local machine. The way to upload the build to the hardware platforms was different for each platform. The exact setup and procedure followed is described in the following segments.

4.4.1 Platform Setup

- TelosB

  Virtual COM port (VCP) drivers were required to enable the device detection as COM ports [48]. Although Ubuntu OS comes with the driver preinstalled, the driver installation was still required on the host machine (Windows 10) to enable the detection of TelosB devices on the virtual machine (Ubuntu). The applications were compiled and flashed (i.e., uploaded to the hardware device) in VMWare for both operating systems.

- IoTLAB-M3

  The process was somewhat different for the IoTLAB testbed. RIOT applications were compiled via vagrant (through SSH), whereas Contiki (version 3.0) applications were compiled on the testbed itself. The files required to compile the applications for IoTLAB-M3 platform were available only on the testbed itself. The process to set up the operating system on the testbed can be found in [50]. RIOT build files were uploaded to the IoTLAB-M3 nodes via the testbed's user interface, whereas Contiki build files were uploaded to the IoTLAB-M3 nodes via command line [51].

4.4.2 Data Logging

Data logging was another aspect that required setup. It was done differently for each platform. "Serial Aggregator" [7] was used for logging data from the IoT-Lab testbed whereas a python script was used to capture serial data generated by TelosB motes. Python scripts were also used to analyze the data logs and to generate graphs.

4.5. Experiment Design

The design of the experiments was simple. Depending on the type of experiment, we either selected two or six nodes. The term "node" here simply represents a hardware device. The selected nodes were always adjacent. Experiments related to the interference had six nodes. Different experiments used different nodes on the testbed depending on what was available at the time of experiment. All the nodes selected were stationary, and the selection was not always same. This is because IoTLAB is an open testbed and nodes were selected depending upon their availability. But according to the IoTLAB administrators, all the nodes are identical and have the exact same configuration. Similarly, for experiments performed without interference, we limited the selected nodes to two. Figure 7 shows a layout of IoTLAB-M3 nodes during an experiment.

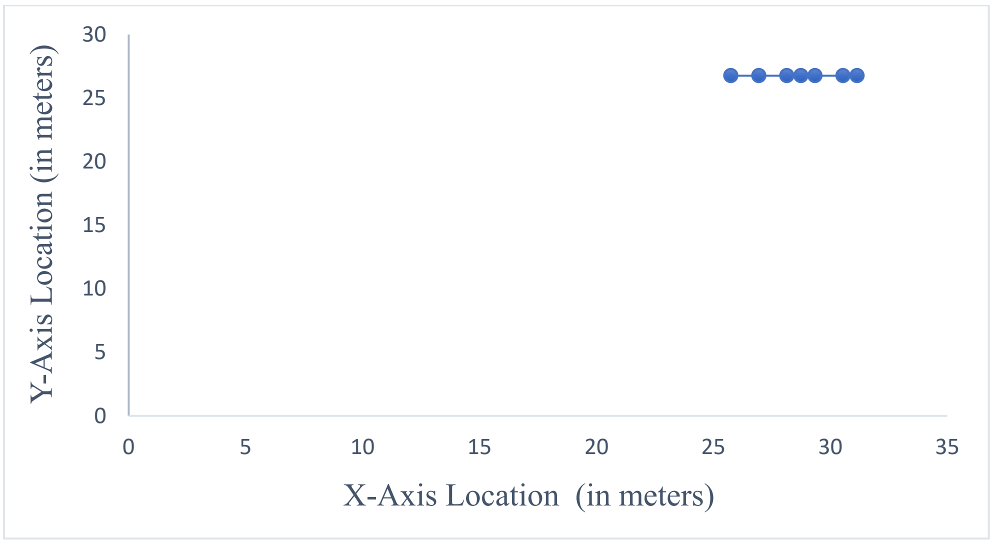

Figure 7. Layout of IoTLAB-M3 node during an experiment.

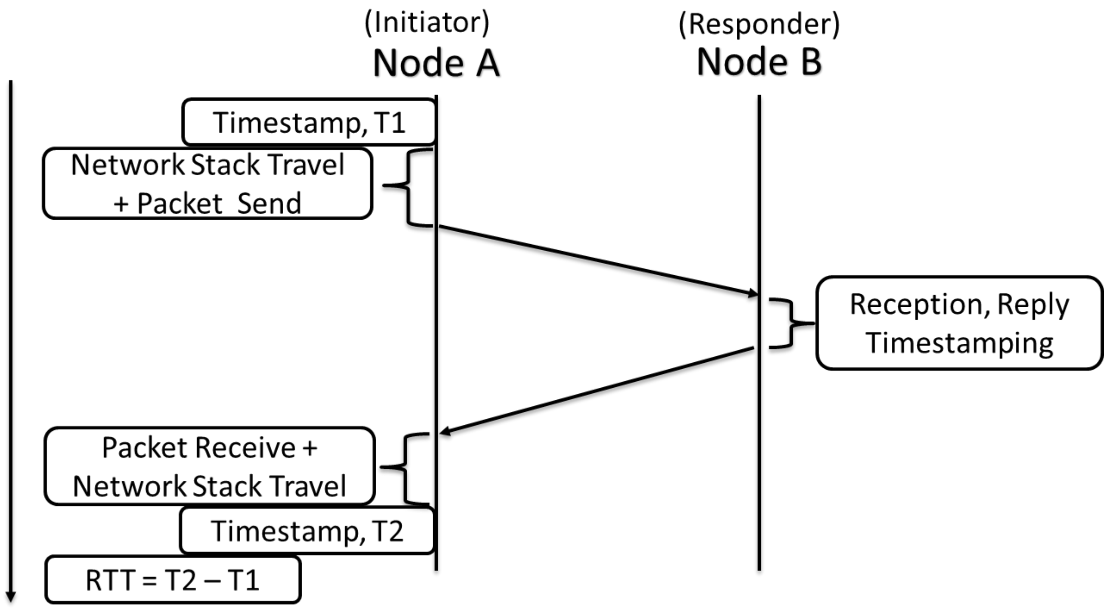
(Initiator)
Node A
(Responder)
Node B
Timestamp, T1
Network Stack Travel + Packet Send
Reception, Reply Timestamping
Packet Receive + Network Stack Travel
Timestamp, T2
RTT = T2 – T1

Figure 8: Timing and Flow of Various Components in an RTT Measurement Experiment.

Figure 8 shows the timing of different operations during an experiment. The experiments without interference were performed with only two nodes. For the sake of description, let's assume that Nodes A and B in Figure 8 are the two nodes from the experiment. Node A would send a packet to Node B, and upon reception, Node B would send a packet back to Node A. We timestamped Node A twice, right before sending the packet (to Node B) and right after receiving the packet (from Node B). The difference between timestamps T2 and T1 represents the round-trip-time which was the primary focal point for jitter analysis. We repeated this experiment for 1,000 packets for IoTLAB-M3 and for 100 packets for TelosB.

Apart from the round-trip-time calculation, we performed other analysis as well, represented as "Network Stack Travel', "Packet Send", and "Packet Receive" in Figure 8. These represent delay analysis when a packet travels from the application layer to the driver layer module and vice versa. And "Reception, Reply Timestamping" simply denotes delay analysis performed at the responder node (Node B).

We evaluated jitters with two kinds of packets, the MAC packet and the UDP packet. MAC and UDP represent two layers in the network stack. Any network stack will incorporate these two protocol layers, although their implementation may differ. Medium Access Layer (MAC) is responsible for the transmission of data packets to and from the lower layer. MAC represents the link layer in the networking stack and is capable of packet retransmission in case of congestion detection. UDP, which stands for User Datagram Protocol, is an alternative to TCP (Transmission Control Protocol) that represents transport layer in the networking stack. It avoids additional processing such as error correction and packet retransmission to allow faster communication.

4.6. Experiment Summary

The following tables (Table 1, and Table 2) summarizes all the experiments performed during this thesis. The table describes the name of the experiment, the hardware platform, the operating system used and a short description about the experiment.

Table 1. Summary of RTT Measurement Experiments

Experiment Name	Hardware Platform	Operating System	Description
Unicast with Interference	IoTLAB-M3	RIOT	This experiment was performed using UDP and MAC protocols in presence of interfering nodes. The experiments were performed with varying rate of interference.
Unicast with Interference and varying queue size	IoTLAB-M3	RIOT	This experiment was performed using UDP and MAC protocols in presence of constant interference rate. This experiment tested the jitter performance due to changes in message queue size.

Table 1 cont.

Unicast without Interference	IoTLAB-M3/ TelosB	RIOT/Contiki	This experiment was performed to test jitters caused by the operating system by removing the presence of interference and by keeping the data rate low.
Analysis of Delay in the Network Stack	IoTLAB-M3	RIOT/Contiki	This experiment was performed to analyze the delay in the networking stacks. Packets were transferred using UDP.
Analysis of Delay at the Driver/Radio Layer	IoTLAB-M3/ TelosB	RIOT	This experiment was performed to find the jitters caused by driver level modules. Packets were transferred using MAC.

Table 2. Summary of Other Experiments

Experiment Name	Hardware Platform	Operating System	Description
Serial Aggregator – Print Test (RIOT Applications)	IoTLAB-M3	RIOT	This experiment was performed to test the timestamping accuracy of IoTLAB's serial data logger, "Serial Aggregator". The timestamping ability was compared with RIOT built-in timer module, "xtimer".
Serial Aggregator – Broadcast Test (RIOT Applications)	IoTLAB-M3	RIOT	This experiment was performed to test the timestamping accuracy of "Serial Aggregator" vs. "xtimer". In this experiment, packets were broadcasted, and reception times were measured.
Serial Aggregator – RTT Test (RIOT Applications)	IoTLAB-M3	RIOT	This experiment also tested the accuracy of the serial aggregator's timestamping. A unicast packet transfer was done for 1,000 packets, and round-trip-time (RTT) was calculated using xtimer's timestamp and serial aggregator's timestamp.

4.7. Results

All the results for applications built with RIOT and evaluated on IoTLAB-M3 have been obtained for 1,000 packets. Rest of the results were obtained for 100 packets only.

The following table summarizes the results of all the experiments performed for this thesis. The table describes the name of the experiment, median round-trip-time, and jitters observed. A section provides additional comments about the results, if any. A detailed summary of the experiments can be found in Table 1.

Table 3. Result Summary

Experiment	Delay (Median)	Jitters	Comment
Unicast with Interference	–	10 to 40 ms	RTT
Unicast with Interference and varying queue size	–	8 to 35 ms	RTT
(IoTLAB-M3/RIOT) Unicast without Interference	14 ms	4.5 ms	RTT
(TelosB/RIOT) Unicast without Interference	24 ms	160 µs	RTT
(IoTLAB-M3/Contiki) Unicast without Interference	6.35 ms	~ 100 µs	RTT

Table 3 cont.

(IoTLAB-M3/RIOT) Analysis of Delay in Network Stack	–	Negligible	This wasn't the major source of jitters
(IoTLAB-M3/RIOT) Analysis of Delay in Driver Layer	6 ms	~ 4.5 ms	Major Jitter Source
(TelosB/RIOT) Analysis of Delay in Driver Layer	17 ms	~ 160 µs	Major Jitter Source
(IoTLAB-M3/Contiki) Analysis of Delay in Network Stack (Sender and Receiver)	1.86 ms (each)	~ 50 µs (each)	Major jitter source
(IoTLAB-M3/Contiki) Analysis of Delay in Driver Layer	1.8 ms	~ 20 µs	Minor jitter source

4.7.1. RTT with Interference

In this experiment, we studied, how the presence of other packets in the network (busy channel due to interference from other packets) impacted jitter. We used IoTLAB-M3 and RIOT OS for these experiments. The setup included four nodes broadcasting "unwanted" packets in the network and two that tried to communicate with each other. Primary nodes (Node A and Node B, Figure 8) communicated with each other in the unicast manner, whereas the broadcasting nodes sent packets to every other node except themselves.

Primary nodes communicated at a very low data rate. Next packet was sent only after receiving a reply to the previous packet. We varied the rate of interference in the network by changing the rate at which the interfering nodes sent the broadcast packets. We modified the rate of broadcasting by altering the sleep duration between packets from 200000 µs to 1s. This allowed us to create an environment with varying levels of interference. The results confirmed that interference is indeed one of the causes of jitters in the network. The jitters increased from 10 ms to about 35 ms as the interference rate increased from 0 packets per second to 20 packets per second for UDP packets. Jitters increased from 10 ms to 40 ms for MAC packets for a similar range of interference. For the next two experiments the interference level was kept constant at four packets per second.

Next, we performed an experiment with different queue sizes. We changed the queue size from 8 to 128 for both UDP and MAC packets. We observed that queue size is inversely proportional to jitters as jitters increased from 8 ms to about 35 ms. We observed about 1% packet loss due to the lower queue size.

We also performed an experiment to determine jitters with varying payload size. This experiment was inconclusive. As expected, we noticed that the round-trip-time delay increased from 15 ms (payload size – 59 bytes) to 42 ms (payload size – 219 bytes) for UDP packets.

4.7.2. RTT Measurement without Interference

Since our goal was to examine the OS for any causes of jitter, it was imperative that we carried out future experiments at a very low rate. To ensure this, we held off the second packet until the first packet was completely received. This experiment was performed only with the UDP packet because it allowed us to examine most of the network stack.

- RTT Measurement | Platform – IoTLAB-M3 | OS – RIOT

  We calculated round-trip-time without any interference and observed a jitter of about 4.5 ms (Figure 9). UDP packets were used for this experiment. We also timestamped some components of the MAC layer and calculated the round-trip-time using the data captured at the MAC layer (Figure 10). The delays and jitters in both figures were very similar, and the presence of jitter was very less between the MAC and the application layer. This propelled us to inspect more deeply, so we traced the packet in the GNRC network stack and performed instrumentation at the driver layer.

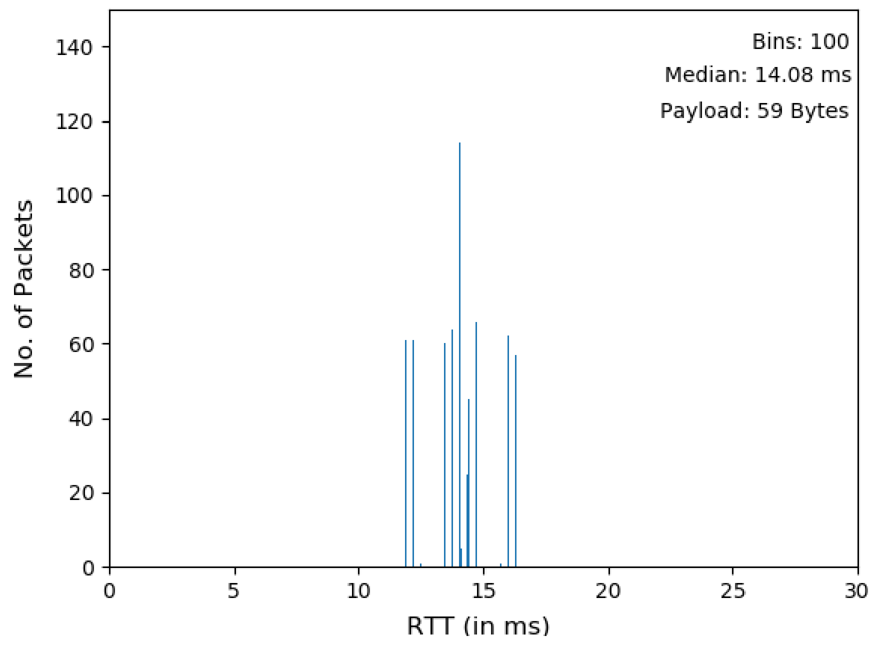

Figure 9. RTT Measured at the Application Layer (IoTLAB-M3, RIOT).

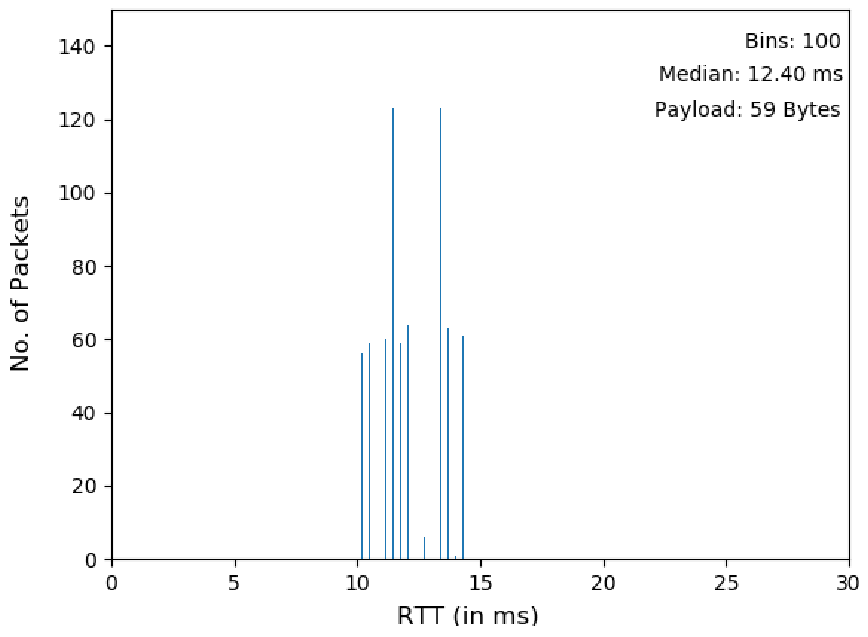

Figure 10. RTT Measured at the MAC Layer (IoTLAB-M3, RIOT).

- RTT Measurement | Platform – TelosB | OS – RIOT

In this experiment, we calculated round-trip-time over the MAC layer. We observed minimal jitters in this case, in the order of few hundred microseconds. Figure 11 shows a jitter of approximately 160 µs.

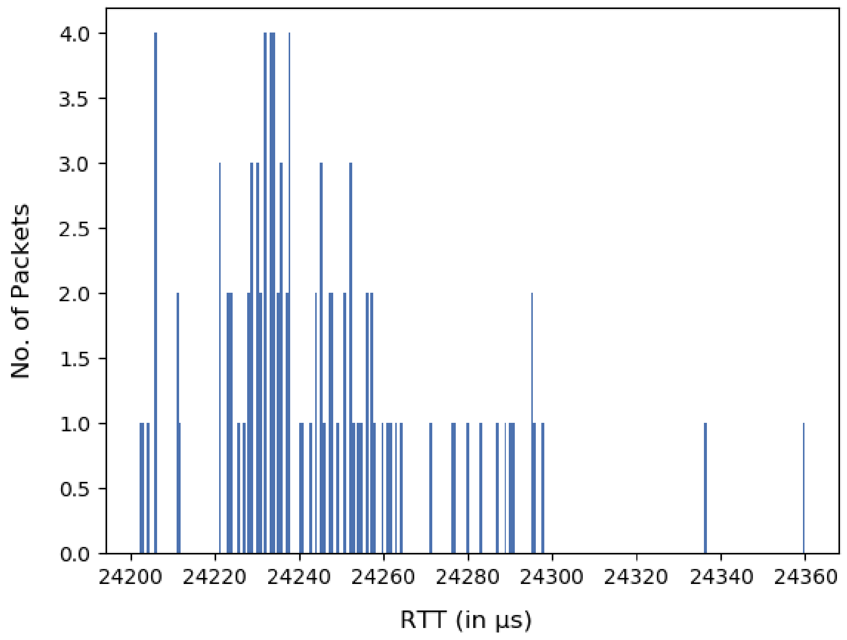

Figure 11. RTT Measured at the Application Layer (TelosB, RIOT).

- Analysis of Delay in the Network Stack | Platform – IoTLAB-M3 | OS – RIOT

In RIOT, each layer in the network stack is a thread. Each thread is continuously looking for any event related to it, via event loop. GNRC works on the basis of message passing (MSG_TYPE_SND and MSG_TYPE_RCV) between layers. Based on the type of message, either "_send" or "_receive" method is invoked. We used these two methods as the point of timestamp capture.

The results for packet trace (Figure 12 and Figure 13) for GNRC network stack processing showed that on the processing front, RIOT has minimal jitter, and a delay of 250-650 μs.

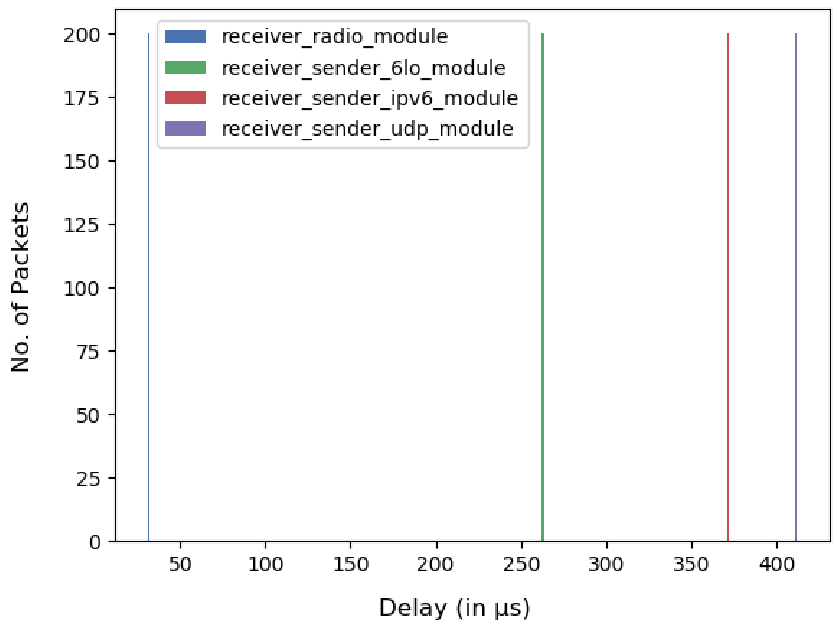

Figure 12. Packet Delays between Networking Modules at the Responder.

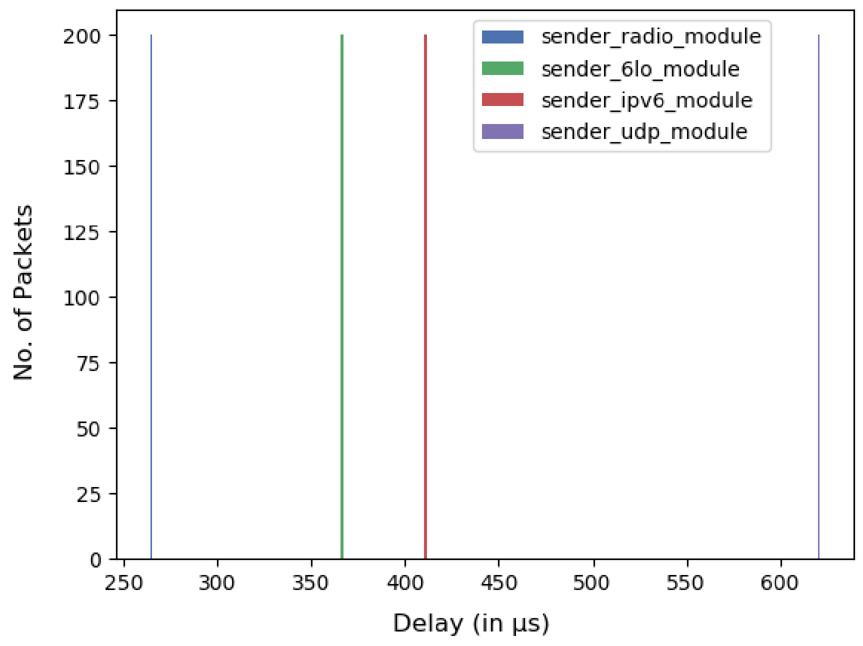

Figure 13. Packet Delays between Networking Modules at the Initiator Node.

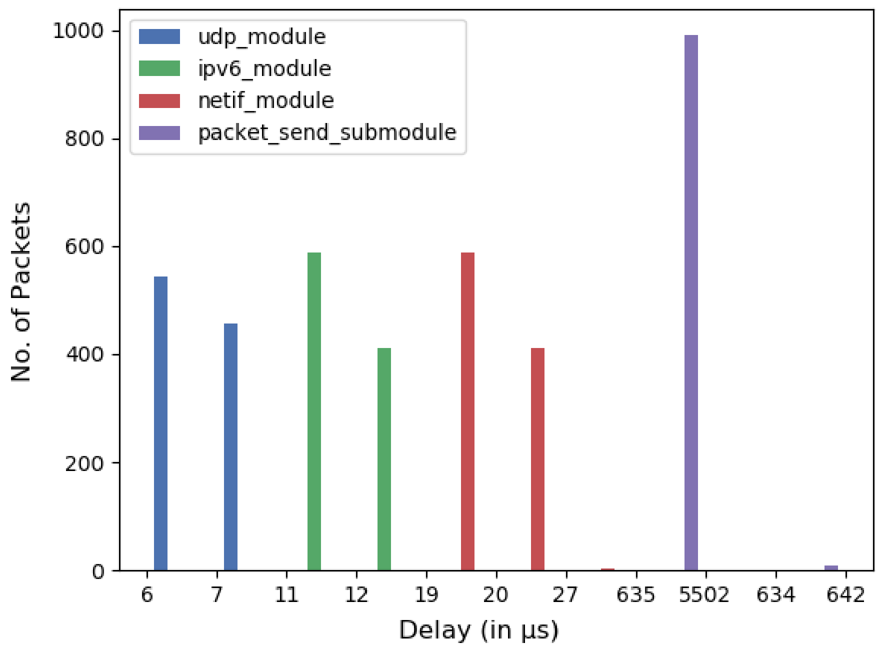

Figure 14. Packet Formatting Delays in Different Networking Modules at the

Initiator Node.

Results for the packet formation process demonstrated that there were some jitters in the packet formation process (Figure 14). But the jitters were very small, in the order of microseconds. Also, since each layer is a thread, we computed run times for each thread (Figure 15). GNRC network stack is optimized such that if no process is happening, the processing switches to IDLE thread. The "Main" and "Idle" threads are not represented in the following graph. Their consumption time forms about 80% of the overall application execution time. Hence, the following run times can be considered accurate and represent the actual processing time for the transmission of packets by the respective network layers.

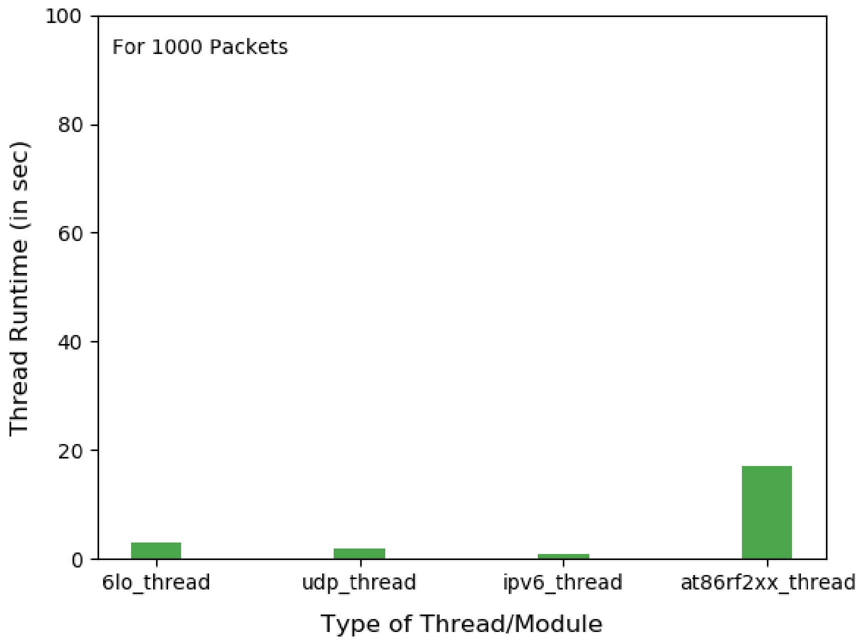

Figure 15. Duration of Thread Runtime at the Initiator Node.

- Analysis of the Delay at the Driver Layer | Platform – IoTLAB-M3 | OS – RIOT

These experiments were performed with a different data rate. We analyzed jitter by maintaining a packet rate of 2.5 packets per second. Even at this data rate, there was no queuing of the packets. The first experiment we performed at driver level was to examine the driver's (at86rf2xx) "send" module. The send module makes call to three submodules namely "at86rfxx_load", "at86rfxx_prepare", and "at86rfxx_exec". Figure 16 represents the delay analysis performed on these sub-modules. The analysis confirmed that these submodules are almost jitter-free.

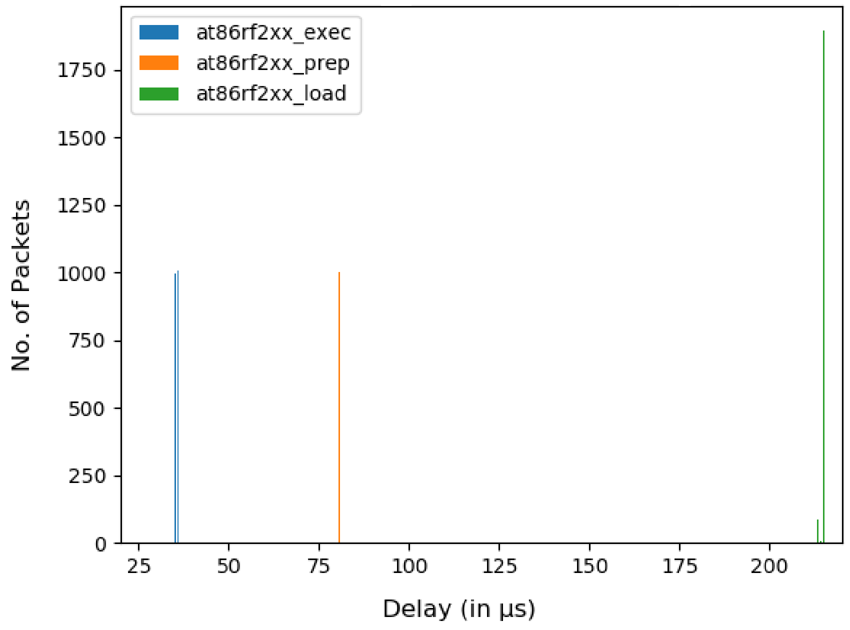

Figure 16. Delays in RTT Due to Driver's Send Submodules.

Next, we timestamped the send sub-module of the driver module (at86rf2xx) and the "irq_handler" (responsible for handling the generated interrupt) of the driver module. End of the "send" module prepares the packet and loads the data into the frame, which is then ready for transmission. And at that point, an Interrupt is generated to inform the CPU about the transmission. We observed, this module (radio module) caused jitters in the round-trip-time.

Table 4. Delays Between Driver Module and Interrupt Service Routine (ISR)

Node	**Mode**	**Time**	**Jitters**
Both	Transmission	6 ms ± 2 ms (Figure 20)	4.5 ms
Receiver	Reception	1.65 ms ± 0.01 ms	0.02 ms
Sender	Reception	0.8 ms	~ 0 ms

We observed a jitter of about 2 ms at both sender and receiver ends (Figure 17). This justified a total jitter of about 4.5 ms from earlier experiments (Unicast without interference for IoTLAB-M3, Figure 9). A similar experiment was also performed to find the delay between the Interrupt and the application level reception process, which was almost jitter-free (Table 4).

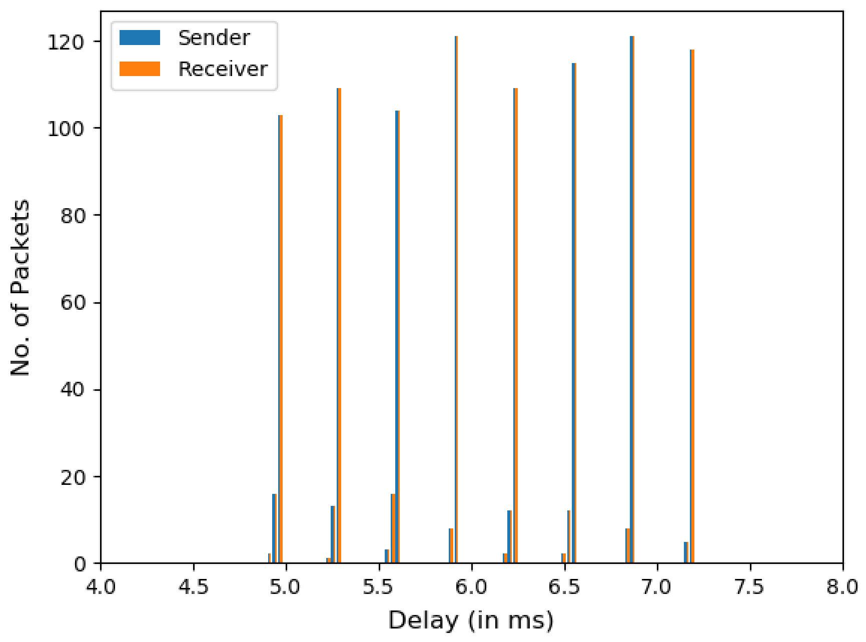

Figure 17. Delays Between MAC Layer and ISR during Packet Transmission.

Next, we performed additional timestamping to ensure that this was the primary cause of jitter. In a key experiment, we took the initiator node jitter out of the equation by capturing the timestamp after the ISR processing was done (Figure 18). We also captured the delay between driver and ISR at the responder end (Figure 19). We observed that jitter measure matched in both experiments, thus proving a correlation between jitter and interrupt handling. This allowed us to assert that the interrupt handling process while transmitting a packet was the cause of the jitter.

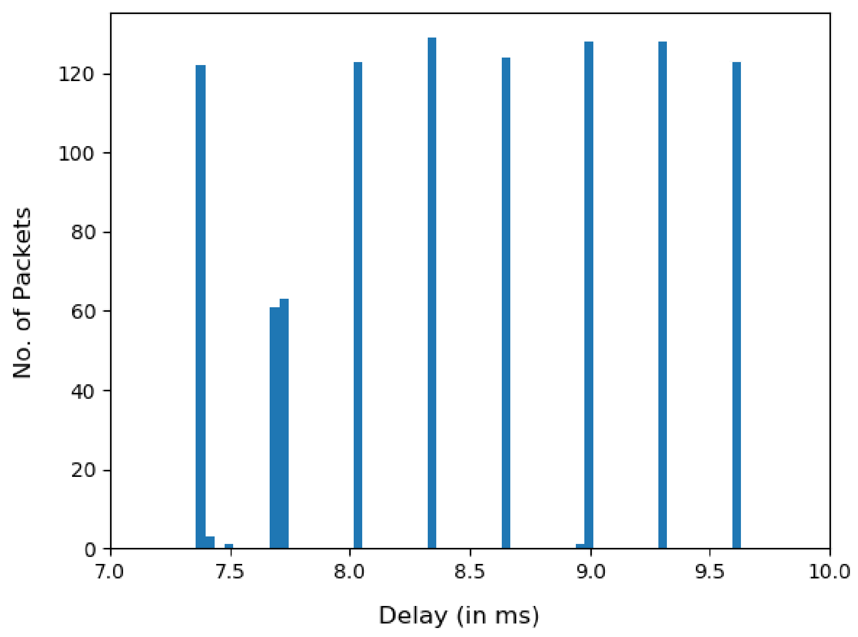

Figure 18. RTT at the Driver Level after Initiator Completed the Transmission.

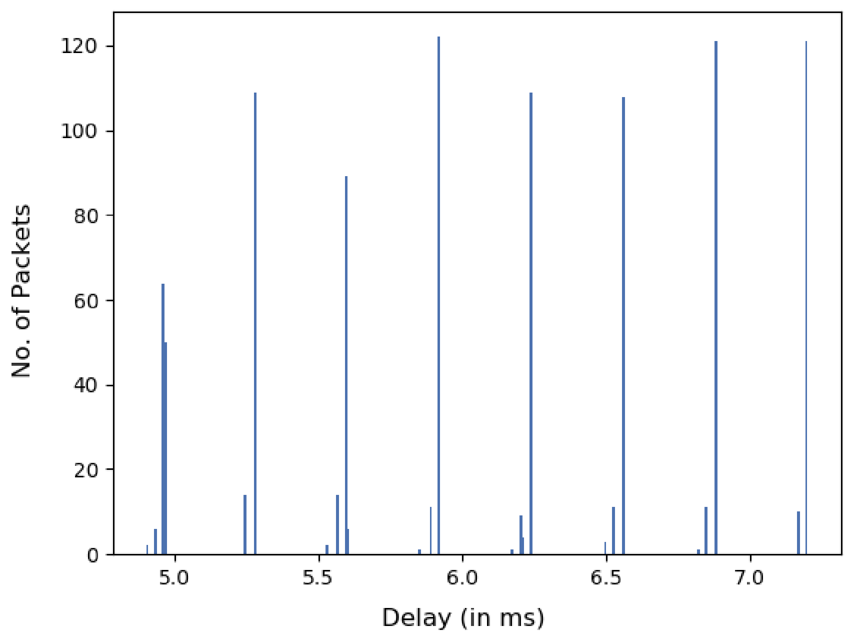

Figure 19. Packet Transmission Time at the Responder Node After Packet Reception

Until Interrupt Invocation.

- Delays in RTT at the Driver Layer | Platform – TelosB | OS – RIOT

For TelosB, we examined the communication process and found that no interrupt was generated during the transmission of a packet. ISR was invoked only during the reception of a packet. Interestingly, in the following code snippet, we can see that the ISR module contains only NETDEV_EVENT_RX_COMPLETE event, unlike the "at86rf2xx" module, which had multiple event-related macros such as NETDEV_EVENT_RX_COMPLETE, NETDEV_EVENT_TX_COMPLETE, etc.

```
static void _isr(netdev_t *netdev)

{

 netdev->event_callback(netdev,NETDEV_EVENT_RX_COMPLETE)

}
```

Combining the fact that there was no interrupt process in the cc2420 driver module for outbound data, and the fact that the jitters for TelosB were in the order of microseconds (Figure 11), we asserted that the major jitter source in RIOT OS was at the driver level. Next, we inspected the sub-components of the driver's send module to identify the cause of small jitter in case of TelosB (Figure 11). The experiment revealed that these subcomponents were almost jitter free (Figure 20).

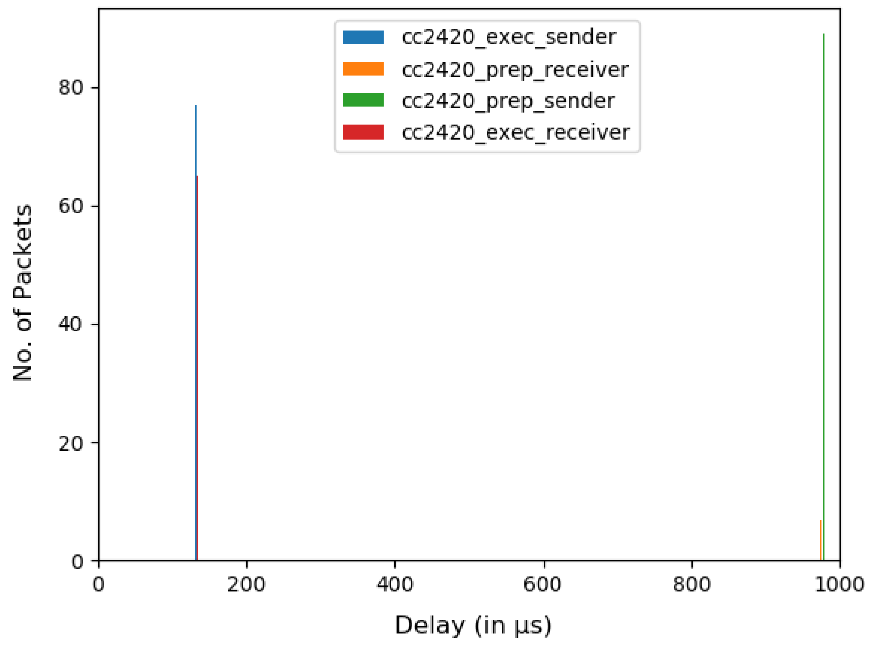

Figure 20. Delays in Submodules of Driver's "send" Module (Initiator and Responder).

Since the driver module was almost jitter-free, we inspected the delays in the sending and reception process, First, we tested the sending process (at initiator node) and calculated delay between the application and the mac layer; this process appeared almost jitter-free (Table 5). Then, we calculated delay between the mac and the driver layer; this process contributed about 25% (~50 µs) to overall jitters (Table 5). Finally, we analyzed jitters at the responder end. We learned that the processing at the responder's end contributes about 50% of the overall jitters (~ 80 µs) (Figure 21).

Table 5. Delays in between RIOT Modules (for TelosB)

Timestamping Locations	**Time**	**Jitters**
Between Application Layer and Mac Module	235 µs	6 µs
Between Mac Layer and Driver Module	950 µs	50 µs

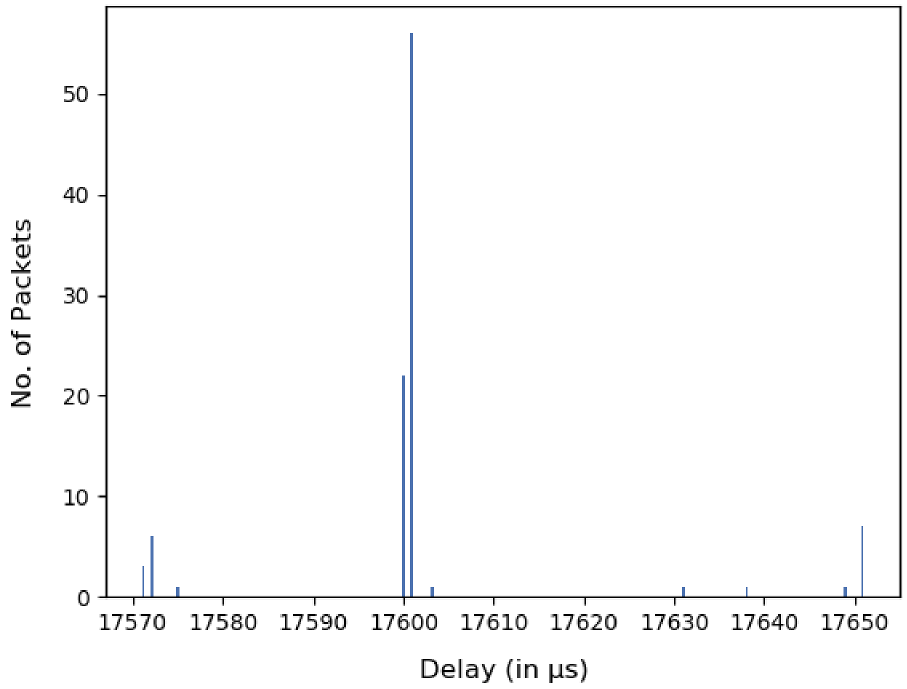

Figure 21. Processing Delay at the Responder Node.

- RTT Measurement | Platform – IoTLAB-M3 | OS – Contiki

  In the Contiki applications, the packets were transferred using broadcast mechanism since we were not able to configure the application to use the unicast mode of transmission. Data rate for experiments under this sub section was one packet per second.

We used uIP networks stack for this experiment, which we simplified by replacing the protocols with the dummy ones. We modified protocols such as MAC, and RDC to 'NULL_MAC', and 'NULL_RDC' respectively. We also switched FRAMER module to 'NULL_FRAMER'. These layers acted as a dummy module i.e. their only function was to receive the packet and forward it to the next layer. These modifications allowed us to remove the processing delay incurred from these layers or modules.

We calculated the round-trip-time without any interference and median round-trip-time came out to be 6.348 ms with a jitter of about 100 μs (Figure 22). UDP packets were used for this experiment. We observed a contrast in the result obtained from a previous experiment performed on the same platform (Figure 9). The reduction in round-trip-time can be attributed towards the modifications we made in the network stack.

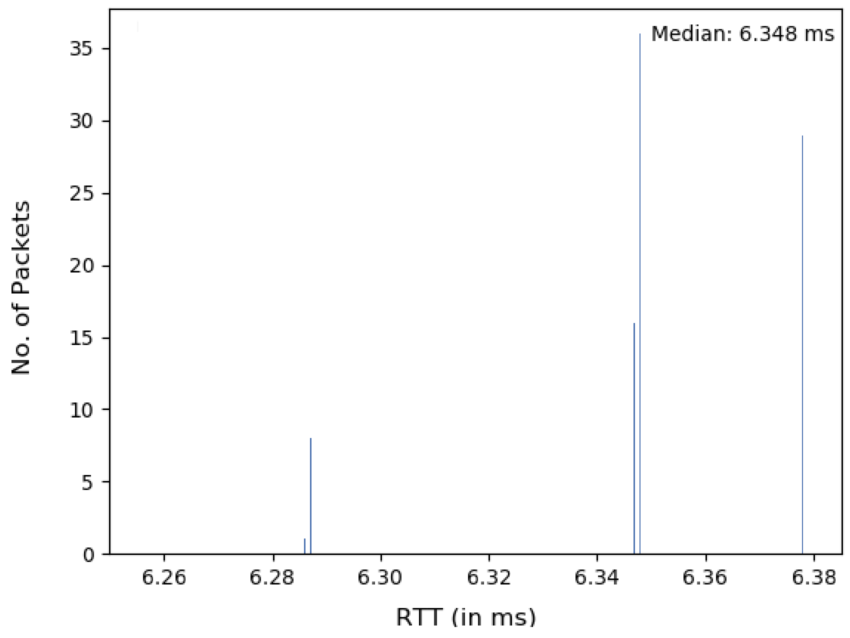

Figure 22. RTT Measured at the Application Layer. (IoTLAB-M3, Contiki).

- Analysis of Delay in the Network Stack | Platform – IoTLAB-M3 | OS – RIOT

In this experiment we performed a deep analysis of the Contiki network stack. We observed that the microseconds of jitters can be attributed to the packet traversal time. Table 6 shows the results obtained from the analysis. Clearly, both sender and receiver were equally responsible for the jitters.

Table 6. Network Stack Traversal Analysis

Timestamping Locations	**Node**	**Time**	**Jitters**
Between UDP Layer and Radio Module	Sender	1.86 ms	~ 60 µs
Between UDP Layer and Radio Module	Receiver	1.86 ms	~ 60 µs

- Analysis of the Delay at the Driver Layer | Platform – IoTLAB-M3 | OS – Contiki

  Next, we performed an experiment to identify the time taken by radio module (driver layer). Driver's "send" module has 2 submodules, "prepare" and "transmit". The results show that the jitter was very small indicating that the waiting period (for transmission to start) was minimal (Table 7).

Table 7. Driver Radio Module Analysis

Timestamping Locations	**Node**	**Time**	**Jitters**
Before and after 'Radio Send' (driver level)	Sender	1.8 ms	20 µs

To compare the interrupt handling process of Contiki with RIOT for the ATMEL radio interface (at86rf231) we had a closer look at the Contiki's driver module implementation for IoTLAB-M3 platform. We noticed, interrupt handling was different in Contiki. Unlike RIOT, Contiki's interrupt handling was synchronous i.e. the "IRQ module" was enabled within the "transmit module". And transmission gets completed before exiting the "transmit module". Whereas in RIOT, at the end of the send module, only the frame became ready to be transmitted. Transmission started after the processing of IRQ handler was completed. Results from these experiments further solidified our conclusion about jitter in RIOT OS.

4.7.3. Evaluation of Time Stamping Accuracy in IoTLAB

- Print Test

In this experiment, we compared the built-in timer module for RIOT (xtimer) with IoTLAB's Timestamping (via serial aggregator script). This experiment was designed to print current time every half-second. This experiment showed that the xtimer module was more accurate when compared to IoTLAB (Figure 23). The median deviation from 0.5 second was 555 µs for xtimer, whereas it was 863.5 µs for IoTLAB. The variation of time deviations (for IoTLAB timestamps) spread from 250 µs to 2400 µs.

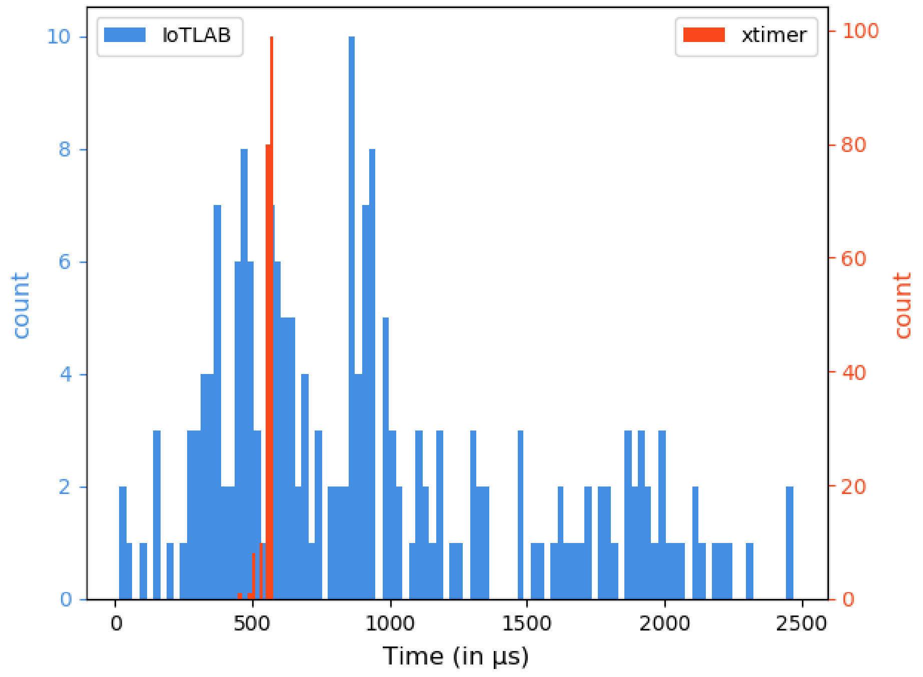

Figure 23. Comparisons of Deviation Produced by RIOT's Clock and IoTLAB Timestamps (in µs, Deviation from 0.5 seconds).

- Broadcast Test

We performed a broadcast experiment with twenty-one IoTLAB-M3 nodes. Amongst these twenty-one, only one node broadcasted whereas the rest only received. The broadcasting node was centrally located to allow the best reception range. Once again, we used the inbuilt timer for RIOT, xtimer, for internal reception timestamp logging and compared serial aggregator timestamp logging against xtimer to identify the difference in reception times. Ideally the receptions would happen very close to each other. As evident in the Figure 25, the reception times in the case for xtimer were much closer to the expected outcome. In the case of serial aggregator, logging was nowhere close to the expected results. Serial aggregator does the logging serially (Figure 24).

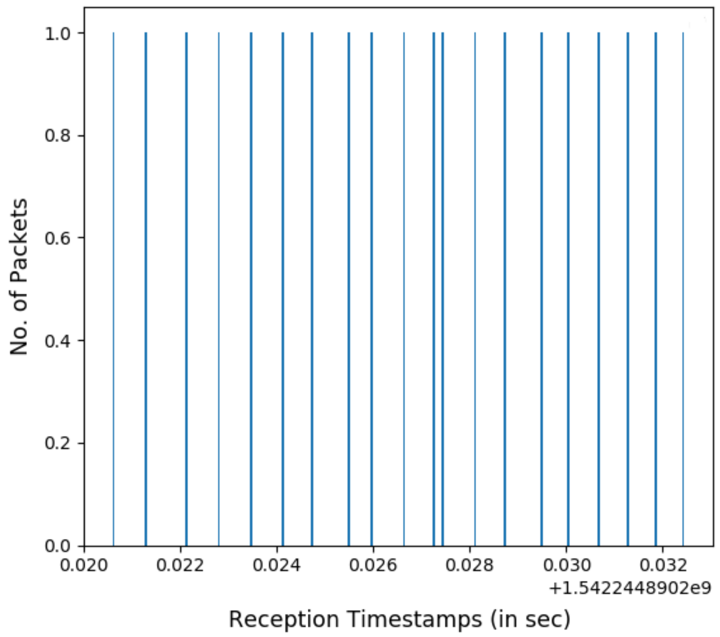

Figure 24. Packet Reception Time for 20 Packets (in Seconds) (IoTLAB).

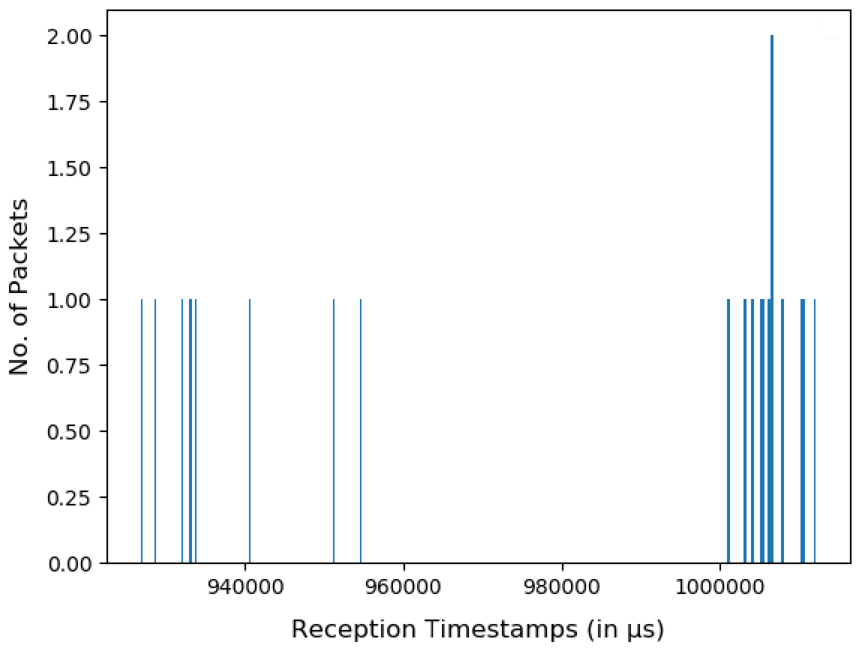

Figure 25. Reception Times for 20 Packets (xtimer).

- RTT Measurements with xtimer vs serial aggregator

Finally, we compared the round-trip-times for a unicast transmission, calculated using xtimer (Figure 27) and IoTLAB (Figure 26). We found less jitter (approximately 4.6 ms vs 6.4 ms) in the case of xtimer. Serial aggregator's timestamping caused an additional jitter of approx. 1.8 ms and skewed the round-trip-time delay by approx. 1.3 ms. Hence, caution should be used when utilizing the timestamps in the logs generated by the serial aggregator.

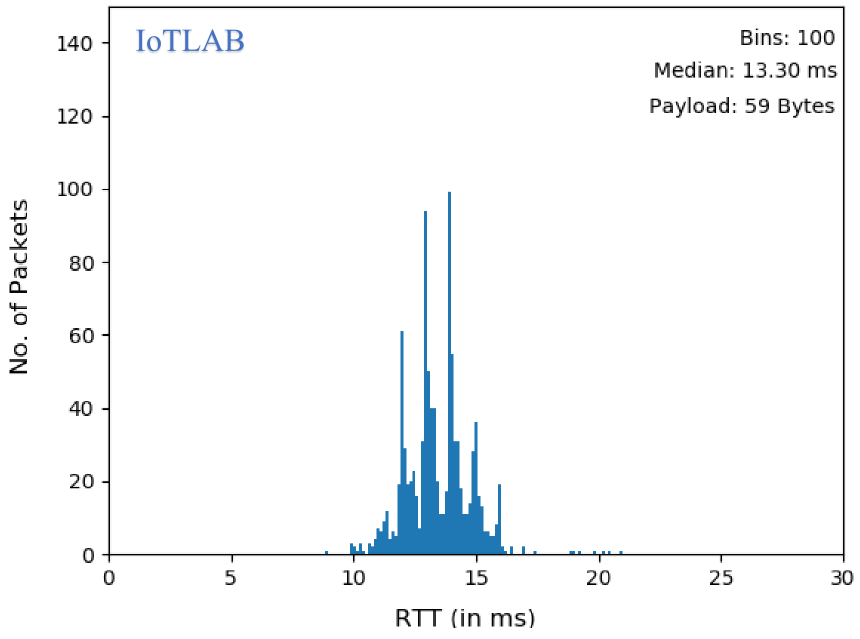

Figure 26. Round-Trip-Time (calculate with IoTLAB).

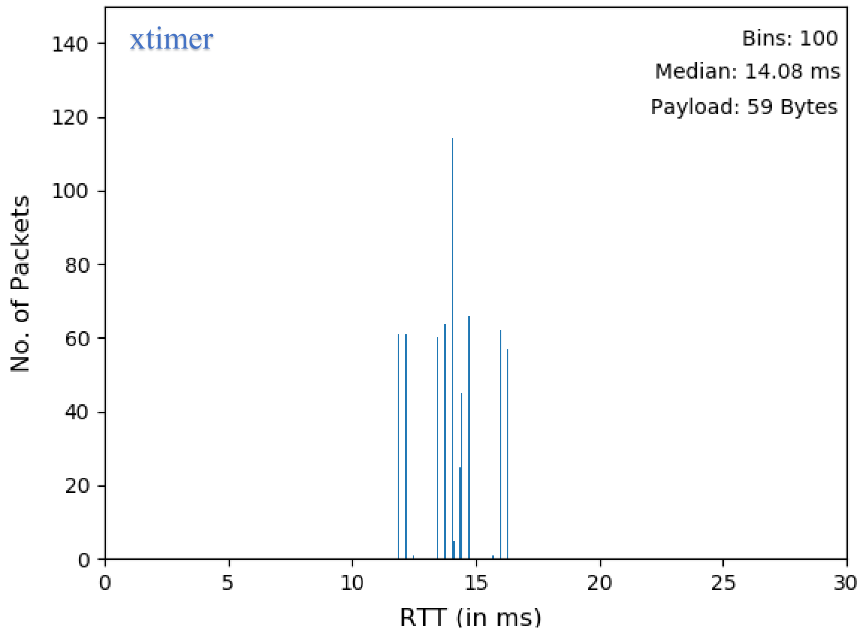

Figure 27. Round-Trip-Time (calculate with xtimer).

Chapter 5

CONCLUSION

This thesis presented a study of network stack performance, layer-wise packet trace, and its analysis, while keeping the focus on identification of jitters in the IoT OS and the contributing factors behind their presence. In this thesis a series of measurement studies of basic applications were performed on IoT hardware and OS platforms. We evaluated this study with two Operating Systems – RIOT and Contiki OS and two IoT hardware platforms – IoTLAB-M3 open node and TelosB.

We identified interrupt handling during packet transmission as the root cause of jitters in the RIOT OS. We tested the RIOT OS with MAC and UDP packet transmissions. Due to resource constraints, only MAC packet transmission was performed on TelosB. The GNRC network stack was consistent with jitters in the order of microseconds, and the lack of interrupt handling for outbound packets in TelosB strengthened our conclusion that, interrupt handling during packet transmission was the root cause of jitters in the RIOT OS. As for Contiki, we were only able to evaluate IoTLAB-M3 platform. We observed that RTT for IoTLAB-M3, measured using RIOT application, was twice as much, when compared to the measurements obtained using Contiki application. The lower jitter in the

Contiki measurements, compared to the RIOT measurements, could be due to the synchronous interrupt handling and simpler network stack used in the Contiki experiments.

As a consequence of the jitter analysis performed on IoTLAB-M3 open node, we identified an issue in the timestamp in the logs produced by serial aggregator (logging script for IoTLAB testbed). We discovered that the timestamps were not accurate and should not be trusted to make time-sensitive computations.

Our experiments with Contiki were not complete and were inconclusive. In the future, we would validate the results for Contiki OS and expand the evaluation to more hardware platforms and operating systems.

Chapter 6

REFERENCES

[1] "Techtarget," [Online]. Available: https://whatis.techtarget.com/definition/IoT-OS-Internet-of-Things-operating-system.

[2] "RIOT-OS," [Online]. Available: https://www.riot-os.org/.

[3] "Contiki-OS," [Online]. Available: http://www.contiki-os.org/.

[4] "FIT Iot-Lab Hardware," [Online]. Available: https://www.iot-lab.info/hardware/m3/.

[5] "TELOSB Datasheet," [Online]. Available: http://www.memsic.com/userfiles/files/Datasheets/WSN/telosb_datasheet.pdf.

[6] "FIT IoTLab," [Online]. Available: https://www.iot-lab.info.

[7] "Serial Aggregator (FIT IoTLAB)," [Online]. Available: https://www.iot-lab.info/tutorials/serial-aggregator/.

[8] N. Mesbahi and H. Dahmouni, "Delay and jitter analysis in LTE networks," in *International Conference on Wireless Networks and Mobile Communications (WINCOM)*, 2016.

[9] J.-C. Bolot, "End-to-End Packet Delay and Loss Behavior in the Internet," in *SIGCOMM*, 1993.

[10] K. Sohraby and A. Privalov, "End-to-end Jitter Analysis in Networks of Periodic Flows," in *Annual Joint Conference of the IEEE Computer and Communications Societies.*, 1999.

[11] H. Chen, *End-to-end Delay Analysis and Measurements in Wireless Sensor,* MID Sweden University.

[12] M. Karam and F. Tobagi, "Analysis of the delay and jitter of voice traffic over the Internet," in *Proceedings IEEE INFOCOM*, 2001.

[13] L. Z. a. D. X. Li Zheng, "Characteristics of Network Delay and Delay Jitter and its Effect on Voice over IIP (VoIP)," in *IEEE International Conference on Communications*, 2001.

[14] M. I. a. D. A. Rohani Bakar, "Performance Measurement of VoIP over WiMAX 4G Network," in *IEEE 8th International Colloquium on Signal Processing and its Applications*, 2012.

[15] W. J. L. V. a. L. M. Jeffrey De Bruyne, "Measurements and Evaluation of the Network Performance of a Fixed WiMAX System in a Suburban Environment," in *IEEE International Symposium on Wireless Communication Systems*, 2008.

[16] a. L. R. Giorgio Cazzaniga, "Embedded Tools for Network Delay Measurement," in *15th International Telecommunications Network Strategy and Planning Symposium (NETWORKS)*, 2012.

[17] F. G. D. W. a. N. H. Y. Frank Jou, "A Method of Delay and Jitter Measurement in an ATM Network," in *Proceedings of ICC/SUPERCOMM*, 1996.

[18] S. Sarkar and S. S. Prasad, "Packet Delay Analysis in Operational Network," in *Second International conference on Computing, Communication and Networking Technologies*, 2010.

[19] L. Xia, "The Parameters Measurement of the Network Performance," in *Applied Mechanics and Materials*, 2013.

[20] C. Fulton and S.-Q. Li, "Delay Jitter Correlation Analysis for Traffic Transmission on High Speed Networks," in *Proceedings of INFOCOM'95*, 1995.

[21] A. B. a. A. P. Stefano Bregni, "Active Measurement and Time-Domain Characterization of IP Packet Jitter," in *IEEE Latin-American Conference on Communications*, 2009.

[22] B. K. G. S. a. Á. L. Miklós Maróti, "The Flooding Time Synchronization Protocol," in *SenSys '04 Proceedings of the 2nd International Conference on Embedded Networked Sensor Systems*, 2004.

[23] M. Z. L. T. a. O. S. Federico Ferrari, "Efficient Network Flooding and Time Synchronization with Glossy," in *Proceedings of the 10th ACM/IEEE International Conference on Information Processing in Sensor Network*, 2011.

[24] J. Mitaroff-Szécsényi, P. Peter and S. Thilo, "Compensating Software Timestamping Interference from Periodic Non-Interruptable Tasks," in *22nd IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*, 2017.

[25] N. M. Shukeri, "Empirical Testing of Prototype Real-Time Multi-hop MAC for Wireless Sensor Networks," in *6th IEEE International Conference on Control System, Computing and Engineering (ICCSCE)*, 2016.

[26] a. A. C. B. Lincoln, "JITTERBUG: a Tool for Analysis of Real-Time Control Performance," in *Proceedings of the 41st IEEE Conference on Decision and Control*, 2002.

[27] C.-C. T. a. W.-C. X. Kuan-Ta Chen, "OneClick: A Framework for Measuring Network Quality of Experience," in *2009*, IEEE INFOCOM.

[28] D. C. L. F. a. G. M. Leopoldo Angrisani, "Type A Uncertainty in Jitter Measurements in Communication Networks," in *IEEE International Instrumentation and Measurement Technology Conference*, 2011.

[29] D. C. L. F. a. G. M. Leopoldo Angrisani, "Packet Jtter Measurement in Communication Networks: aSensitivity Analysis," in *IEEE International Workshop on Measurements and Networking Proceedings (M&N)*, 2011.

[30] H. G. X. Z. C. L. a. Y.-a. T. Quanxin Zhang, "A Sensitive Network Jitter Measurement for Covert Timing Channels Over Interactive Traffic," in *Multimedia Tools and Applications*, 2108.

[31] U. Premaratne, "Empirical Network Jitter Measurements for the Simulation of a Networked Control System," in *14th International Conference on Advances in ICT for Emerging Regions (ICTer)*, 2014.

[32] D. T.-R. D. M.-R. a. C. V.-R. L. Rizo-Dominguez, "Jitter in IP networks: A Cauchy Approach," 2010.

[33] E. S. D. B. a. M. R. P. Ferrari, "Evaluation of communication latency in Industrial IoT applications," in *IEEE International Workshop on Measurements and Networking Proceedings (M&N)*, 2011.

[34] N. Khalil, M. R. Abid, D. Benhaddou and M. Gerndt, "Wireless Sensors Networks for Internet of Things," in *IEEE Ninth International Conference on Intelligent Sensors*, 2014.

[35] M. Bermudez-Edo, T. Elsaleh, P. Barnaghi and K. Taylor, "IoT-Lite: A Lightweight Semantic Model for the Internet of Things," in *Intl IEEE Conferences on Ubiquitous Intelligence & Computing*, 2016.

[36] S. D. T. Kelly, N. K. Suryadevara and S. C. Mukhopadhyay, "Towards the Implementation of IoT for Environmental Condition Monitoring in Homes," in *IEEE Sensors Journal Oct. 2013*, October 2013.

[37] G. Z. Papadopoulos, T. Matsui, P. Thubert, G. Texier, T. Watteyne and N. Montavont, "Leapfrog Collaboration: Toward Determinism and Predictability in Industrial-IoT applications," in *IEEE International Conference on Communications (ICC)*, 2017.

[38] A. Dunkels, B. Gronvall and T. Voigt, "Contiki - a Lightweight and Flexible Operating System for Tiny Networked," in *29th Annual IEEE International Conference on Local Computer Networks*, 2004.

[39] "GNRC Packet Reception Process," [Online]. Available: https://riot-os.org/api/group__drivers__netdev__api.html.

[40] "Network Model GNRC (Netdev)," [Online]. Available: https://github.com/RIOT-OS/RIOT/wiki/Model-for-the-network-stack#initialize-network-device.

[41] "IoTLAB_Description," [Online]. Available: https://www.iot-lab.info/what-is-iot-lab/.

[42] "FIT IoTLAB M3," [Online]. Available: https://github.com/iot-lab/iot-lab/wiki/Hardware-M3-node.

[43] "TelosB RIOT," [Online]. Available: http://doc.riot-os.org/group__boards__telosb.html.

[44] M. Lenders, P. Kietzmann, O. Hahm, H. Petersen, C. Gündoğan, E. Baccelli, K. Schleiser, T. C. Schmidt and M. Wählisch, "Connecting theWorld of Embedded Mobiles: The RIOT Approach to Ubiquitous Networking for the IoT," *CoRR,* vol. abs/1801.02833, 2018.

[45] "Vagrant Setup - Contiki-NG," [Online]. Available: https://github.com/contiki-ng/contiki-ng/wiki/Vagrant.

[46] "Vagrant Setup - RIOT," [Online]. Available: https://github.com/RIOT-OS/RIOT/tree/master/dist/tools/vagrant.

[47] I. SSH. [Online]. Available: https://www.iot-lab.info/tutorials/ssh-access/.

[48] "FTDI Driver," [Online]. Available: https://www.ftdichip.com/Drivers/VCP.htm.

[49] "Contiki-NG," [Online]. Available: https://github.com/contiki-ng/contiki-ng.

[50] "IoTLAB - Contiki Compilation," [Online]. Available: https://www.iot-lab.info/tutorials/contiki-compilation/.

[51] "IoTLAB CLI Tools," [Online]. Available: https://github.com/iot-lab/cli-tools.

[52] "Techopedia," [Online]. Available: https://www.techopedia.com.

[53] "Wikipedia," [Online]. Available: https://en.wikipedia.org.