Hazra, S., Duquennoy, S., Wang, P., Voigt, T., Lu, C. y Cederholm, D. (2019). Handling Inherent Delays in Virtual IoT Gateways. In *2019 15th International Conference on Distributed Computing in Sensor Systems (DCOSS)*.

DOI: https://doi.org/10.1109/DCOSS.2019.00031

Handling Inherent Delays in Virtual IoT Gateways

Saptarshi Hazra, Simon Duquennoy, Thiemo Voigt
RISE SICS
Stockholm, Sweden
{firstname.lastname}@ri.se

Peng Wang
KTH
Stockholm, Sweden
pengwang@kth.se

Chenguang Lu, Daniel Cederholm
Ericsson Research
Stockholm, Sweden
{firstname.lastname}@ericsson.com

***Abstract*—Massive deployment of diverse ultra-low power wireless devices in different application areas has given rise to a plethora of heterogeneous architectures and communication protocols. It is challenging to provide convergent access to these miscellaneous collections of communicating devices. In this paper, we propose VGATE, an edge-based virtualized IoT gateway for bringing these devices together in a single framework using SDRs as technology-agnostic radioheads. SDR platforms, however, suffer from large unpredictable delays. We design a GNU Radio-based IEEE 802.15.4 experimental setup using LimeSDR, where the data path is time-stamped at various points of interest to get a comprehensive understanding of the characteristics of the delays. Our analysis shows that GNU Radio processing and LimeSDR buffering delays are the major delays. We decrease the LimeSDR buffering delay by decreasing the USB transfer size but show that this comes at the cost of increased processing overhead. We modify the USB transfer packet size to investigate which USB transfer size provides the best balance between buffering delay and processing overhead across two different host computers. Our experiments show that for the best measured configuration the mean and jitter of latency decreases by 37% and 40% respectively for the host computer with higher processing resources. We also show that the throughput is not affected by these modifications.**

***Keywords*-Software Radio; Edge Computing; Internet of Things; RAN Virtualization; IEEE 802.15.4**

I. Introduction

The Internet of Things (IoT) enables communication among huge numbers of diverse low-power devices. According to an estimate by Ericsson [1], there will be 20 billion connected devices by 2023 used in a wide variety of applications like healthcare [2], smart cities [3], smart industries [4] and environmental monitoring [5]. The rapid growth of these solutions has led to a plethora of different heterogeneous architectures and protocols solving specific use cases [6]. This results in the fragmentation of the protocol space with a wide range of options available like IEEE 802.15.4, Wireless Local Area Network (WLAN), Bluetooth Low Energy (BLE) and Narrow Band IoT (NB-IoT). Future application areas like smart industry and smart cities will need to provide convergent access to this miscellaneous collection of communication protocols at the IoT gateway to satisfy different application demands.

Previous approaches [7], [8], [9], [10], [11] try to support these diverse protocols by incorporating dedicated radio hardware for each of the supported communication protocols. As the number of protocols increases, scaling and upgrading these systems become inconvenient. Furthermore, the use of specialized hardware makes it difficult to future-proof these systems.

In this paper, we propose VGATE – Virtualized Gateway, an edge-based IoT gateway architecture, providing convergent access to the wireless medium using Software Defined Radio (SDR). SDR implements the physical layer (PHY) for the communication protocols in software which enables experimentation with the physical layer protocols. Future revisions can easily be incorporated by software updates, making it possible to future-proof the system. This, however, comes at the cost of additional latency and unpredictability of execution times as we move the processing from specialized hardware to general purpose software systems. Previous studies [12], [13], [14] have highlighted that these additional delays are quite significant and result in interoperability problems with dedicated commercial off-the-shelf (COTS) platforms. A comprehensive characterization of these delays is, however, missing. Hence, in this paper, we develop an understanding of the characteristics of these delays. Finally, we propose a solution to reduce these delays.

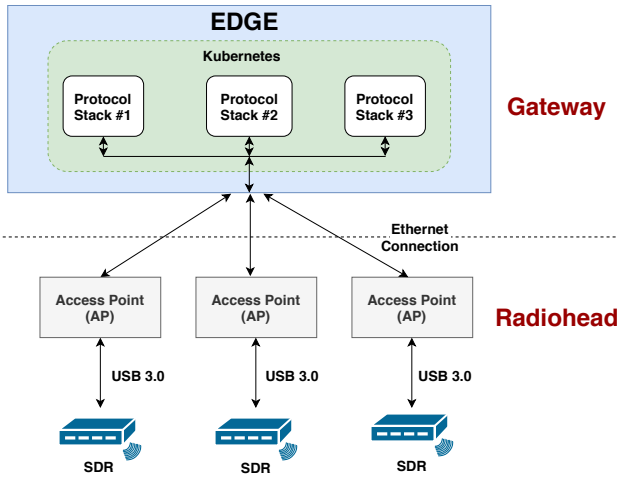

Fig. 1. VGATE: Edge based virtualized RAN architecture for IoT Gateways

We make the following contributions in this paper:

- We propose VGATE, an edge-based virtualized Radio Access Network (RAN) architecture for IoT gateways. It makes IoT deployment more flexible by bringing together multiple protocol stacks in a single framework.

- We evaluate the delays associated with our architecture. For our measurements, we use a LimeSDR-based IEEE 802.15.4 experimental setup. Our measurements show that the LimeSDR buffering delay and GNU Radio processing time are the major bottlenecks in our architecture.
- We address the LimeSDR buffering delay by reducing the USB transfer packet size. Our results show that we achieve a latency improvement of approximately 37% while we decrease the standard deviation associated with the latency measurements by 40%.

The paper proceeds as follows: Section II introduces the VRAN architecture and the need for delay analysis. Our design choices and implementation are described in Section III. Section IV describes our experimental setup and presents our results. We describe related work in Section V. Finally, we present concluding remarks in Section VI.

II. VGATE ARCHITECTURE

In this section, we discuss our VGATE architecture and the need for timing characterization in VGATE.

Fig. 1 shows the VGATE architecture. It is divided into two halves: *gateway* and *radiohead*. The *radiohead* is responsible for managing access and transferring data to and from the wireless medium. The *gateway* hosts the core signal processing processes of different communication protocols. The *radiohead* components and the *gateway* processes are configured and managed by a central control entity which builds a global view of the entire system.

Most of the communication protocols for connected devices such as IEEE 802.15.4, LoRaWAN, BLE and Wi-Fi use the ISM band located around 2.4 GHz and 868 MHz. The spectral coexistence of these protocols can be leveraged to design an access point that provides convergent radio access for these protocols. In our architecture, we have multiple access points sampling the radio spectrum at different frequency bands and different geographic locations using SDRs. This helps to unify the radio channel access for an application area into a single architecture.

In order to ensure a low end-to-end latency between the nodes and the gateway, the transport delay between the *gateway* and *radiohead* should be kept minimal. We use the edge of the network for deployment of *gateway* processes as it enables low latency transport as compared to a cloud-based architecture [15]. Incorporating the *gateway* functionality at the access point will provide the lowest latency, but centralization at the edge provides scalability and flexibility of deployment. The *gateway* hosts the complete software protocol stacks for different supported protocols and can be scaled by adding more software stacks. This allows experimentation and future-proofs the system by enabling software updates of the protocol stacks all the way down to the PHY layer.

The protocol stacks are deployed as own self-contained containers which are managed by a Kubernetes orchestrator. The central control entity manages a logical view of the architecture, whereas the Kubernetes orchestrator manages the actual mapping of these protocol stacks to computing resources. We use ethernet connections between the *gateway* and *radiohead* of the VGATE architecture since there exists a mature collection of transport protocols over ethernet.

Before being deployable, our architecture still has to overcome some challenges, in particular, the delays and jitter imposed by the use of SDR to design the physical layer in software. These delays make it difficult to meet the round-trip times defined in the protocol specification of some protocols such as IEEE 802.15.4. To get a basic understanding of the delays, we evaluate the latency of a standalone implementation using IEEE 802.15.4 at 2.4 GHz as our representative IoT protocol. We use the standalone setup shown in Fig. 3. In this setup, a single host computer is connected to an SDR platform. Fig. 2 presents these baseline measurements.

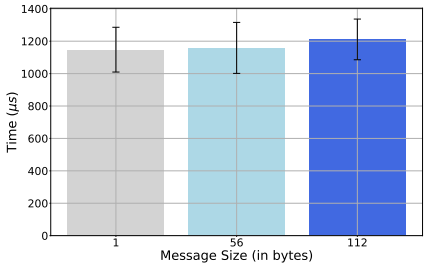

Fig. 2. **Baseline Measurements**: Latency measurements are shown along the vertical axis, with the MAC data payload size shown along the horizontal axis.

We encode a message from the host computer using the IEEE 802.15.4 MAC and PHY. This message is loopbacked by the SDR RF front end. Finally, the message is decoded on the same computer. We define the round trip time for a message as latency. Our experiments show that the lowest latency is $1165\mu s$, which is more than five times the $198\mu s$ specified by the IEEE 802.15.4 protocol for the acknowledgment of data packets. For practical deployments of VGATE we need to understand the cause of these delays and how to minimize them. In this paper, we concentrate on these two questions as they are central to determining the applicability of VGATE for different IoT protocols.

III. DESIGN & IMPLEMENTATION

The major delays in VGATE can be decomposed into two main components: Transport delay over ethernet connections and delays associated with SDR implementation of the physical layer. Chang et al. [16] show the possibility of transport delays in the order of microseconds over ethernet using UDP and raw ethernet sockets. Montarezi et al. [17] and Miao et al. [18] present ethernet transport delays of less than 100 μs by using user space network stacks. In this paper, we focus on the delays associated with the SDR as these previous approaches demonstrate the feasibility of low latency data transfer over ethernet connections. In order to investigate the SDR delays, we converge the *gateway* and *radiohead* functionality into a standalone implementation using a single protocol stack. We

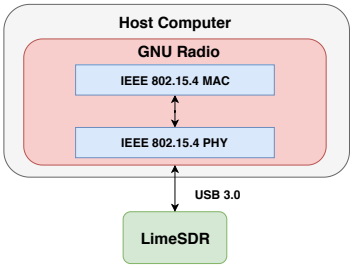

Fig. 3. **Implemented System**: We design a standalone setup, which uses GNU Radio as the digital signal processing framework. The IEEE 802.15.4 PHY and MAC are implemented inside the GNU Radio. The host computer communicates with the LimeSDR over a USB 3.0 connection.

We chose GNU Radio as our software digital signal processing framework as it is the most complete SDR software framework available and is open source, allowing us to modify it according to our needs. As we envision these gateways to be ubiquitous the cost of the SDR platform should be considered. We selected LimeSDR as our platform as it is cost effective as compared to the traditionally used Universal Software Radio Peripheral (USRP) platforms. Furthermore, LimeSDR supports the ISM bands located around 2.4 GHz and 868 MHz.

Fig. 3 shows the overview of our implemented system. The IEEE 802.15.4 Medium Access Control (MAC) and PHY are implemented by adapting the implementation from the WIME [19] project as individual blocks in the GNU Radio. The IEEE 802.15.4 PHY communicates with the LimeSDR over a USB 3.0 bus connection.

IV. Evaluation

A. Experimental Setup

As we are interested in better understanding the reasons behind the delays inherent to our implementation, we decided to ignore the over the air transmission delay. We use a Radio Frequency (RF) loopback configuration on the LimeSDR, where the Transmit (TX) RF front-end path is connected to the Receive (RX) RF path. This configuration uses the complete LimeSDR data path including the digital to analog domain and analog to digital domain conversion. Hence, our measurements will emulate the delays experienced by actual RF transmissions and receptions. In order to test the functional validity of the IEEE 802.15.4 MAC and PHY, we communicate to a Zolertia firefly in the broadcast mode using our setup.

Component Delays This paper defines seven different delays for the purpose of finding performance bottlenecks introduced by different software and hardware components in this implementation. The reasoning behind the choice of delays are as follows:

1) **GNU Radio TX Processing Delay** This provides the delays incurred to complete the signal processing of IEEE 802.15.4 PHY modulation.
2) **LimeSDR TX Driver Delay** The time it takes the driver to pack the data into the data packet structure understood by the LimeSDR FPGA can be an important metric to show if the driver needs a closer look for optimization.
3) **User Space to Kernel Space Delay** This delay highlights the impact of Linux process scheduler and context switching from user space to kernel space on the performance.
4) **LimeSDR loopback Delay** The time taken by the USB 3.0 bus transfer, the buffering in the LimeSDR FIFOs and the hardware processing delays provides an insight on how the transfer of data from the host computer to the LimeSDR and vice versa impacts the performance.
5) **Kernel Space to User Space Delay** It is similar to the user space to the kernel space delay specified earlier. It is measured on the RX path of the setup, whereas user space to kernel space delay is measured on the TX path.
6) **LimeSDR RX Driver Delay** It provides the time needed to unpack the data packets coming from the LimeSDR and forward the samples to the GNU Radio flow graph. It also includes the buffering time in case the GNU Radio RX flow graph is unable to process the samples at the rate they are being generated.
7) **GNU Radio RX Processing Delay** It provides the delays introduced by the RX processing in GNU Radio.

Component delay measurements In order to quantitatively evaluate these delays, we timestamp the experimental setup at different layers of the software process. The timestamp method allows us to measure the actual delays at execution time. This method introduces very low overhead, hence it does not alter the actual measurements which we tested by doing a latency (T_8-T_1) comparison with and without the timestamps.

We define eight different timestamps for the measurements of the component delays mentioned previously. The experimental setup shown in Fig. 4 exhibits the different interacting software layers in our system and the corresponding timestamps for each software layer. We concentrate on finding the last execution statement applied to the data in each software layer using static code analysis. For the TX path, we take the timestamps at the output of each layer as shown in Fig. 4. For the RX path, we take the timestamps at the input of each layer. Note that we measure T_8 when the IEEE 802.15.4 Start of Frame Delimiter (SFD) is detected in the IEEE 802.15.4 packet detector. This allows us to evaluate the latency (T_8 - T_1) as per standard definition.

We define the following relationship between these timestamps and the component delays:

- T_2-T_1: GNU Radio TX Processing Delay
- T_3-T_2: LimeSDR TX Driver Delay
- T_4-T_3: User Space to Kernel Space Delay
- T_5-T_4: LimeSDR loopback Delay
- T_6-T_5: Kernel Space to User Space Delay

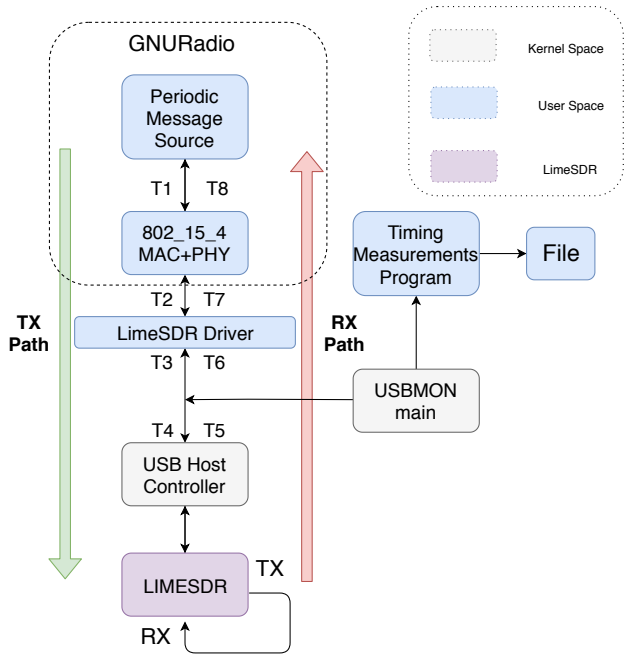

Fig. 4. **Experimental Setup** Overview of various timestamps and their association to the software layers.

- T_7-T_6: LimeSDR RX Driver Delay
- T_8-T_7: GNU Radio RX Processing Delay

In order to ensure that the LimeSDR loopback delay is as accurate as possible to the delay contributed by the buffering in LimeSDR and bus transfer delays, we decided to measure T_4 and T_5 using the timestamps from the USB host controller. We use the USBMon kernel utility to monitor the `urb_submit` and `urb_request` calls to and from the USB host controller. Each USB Request Block (URB) has an associated timestamp which is generated by the kernel USB driver and the USB host controller for `urb_submit` and `urb_request` respectively. Once the events have been received from the USBMon event queue, it is necessary to find the relevant USB transfers in these events. We adopt an offline processing approach, as compared to runtime measurement. Runtime measurements add processing overhead which will impact the component delays. A separate program `Timing Measurement Program` (shown in Fig. 4) collects all the events and writes the relevant samples together with the URB timestamps to a file. We measure the timestamps T_4 and T_5 from this data file after the execution of the experiments.

Since we use analog loopback, the TX sample value changes as it is converted to analog domain and again sampled to digital RX samples. For this reason, the sample values in the TX and RX USB packets cannot be directly compared to find when the same sample is returned. Hence, we use cross-correlation to match the TX and RX USB transfers and find the time shift of the RX samples from the TX samples. The time axis for both the TX and RX samples is enumerated with the URB timestamps of the USB packets. The cross-correlation gives us the URB timestamp for the starting of the relevant samples of the received IEEE 802.15.4 packet, which we define as our T_5. As the TX stream only contains samples from sent TX packets,we use a threshold on the digital sample values to find T_4.

In order to minimize the processing overhead and file operations, we measure T_6 and T_7 for all execution calls in the LimeSDR driver and the GNU Radio. This creates a problem of correlating the timestamps together as there are multiple T_6 and T_7 for a single T_5, T_8 combination. We use the knowledge that the processing time in each of these software layers should be finitely positive, so all the relevant timestamps should follow the relation $T_8 > T_7 > T_6 > T_5$ to find all possible values of T_6 and T_7 which satisfy this condition. We take the first value among these possibility sets as our T_6 and T_7.

Measurement Setup We add a periodic message source in our measurement setup for generating MAC payloads of a specific size, which allows us to evaluate the impact of MAC payload size on the different component delays. The periodicity of the MAC payload generation is set to 500 ms in order to make each measurement independent of the previous. We run our experiments for 500 seconds. This results in 999 messages for measuring the components delays over wider experimental dataset. As the software computation is CPU intensive, we use two host computers in order to evaluate the impact of the host computer processing resources on the characteristics of the component delays. We refer to the two host computers as 'laptop' and 'desktop computer', the hardware specifications for these are shown in Table I. The 'desktop computer' has better processing resources, because of higher CPU clock speed, and two more hardware CPU cores. It uses Ubuntu 16.04.5 LTS as the operating system whereas the laptop uses Elementary OS built on Ubuntu 16.04.5 LTS as its operating system.

TABLE I
HARDWARE SPECIFICATIONS

Resource	Laptop	Desktop Computer
CPU	Intel i5-4300U	Intel i5-3470
CPU Clock Speed	1.9 GHz	3.2 GHz
CPU Cores	2	4
CPU threads	4	4
RAM	7.9 GB	15.6 GB

B. Understanding delays

We performed two experiments for understanding the component delays and the impact of two parameters: MAC data payload size and USB transfer size on these delays. The results and our analysis are summarized below.

Experiment 1:

Motivation The impact of data size on the component delays is necessary to understand how the delays scale if we want to use large packet sizes in our system. This experiment is designed to provide a comprehensive understanding of the component delays and how different MAC payload size and

host computer configuration affect the system performance.

Setup As our IEEE 802.15.4 MAC adds a header of 15 bytes to the data packet, the maximum data payload we can use for a valid IEEE 802.15.4 packet is 112 bytes. We chose three message sizes: 1 byte, 56 bytes and 112 bytes to understand the impact of different MAC payload sizes. We use the same measurement setup described in our experimental setup.

Results We present our results for this experiment in Fig. 5. The results show an increase in TX software delays ($T_4 - T_1$)

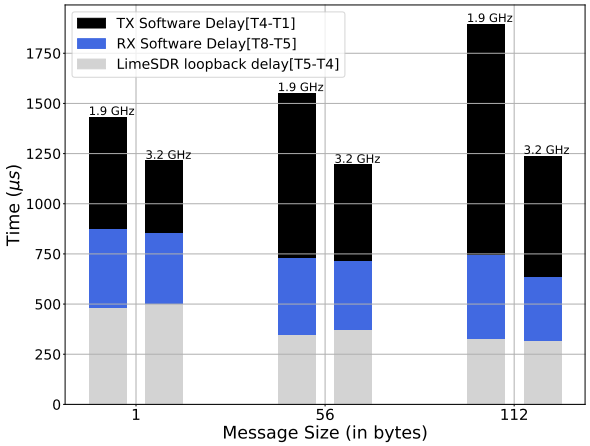

Fig. 5. **Component Delays vs MAC data payload size**: The message sizes are shown along the horizontal axis, with time along the vertical axis. The results for the two host computers for a particular MAC data payload size is shown side by side, with the clock rates identifying the computer used. The component delays have been labeled with different colors. The number of samples in all these results has been set to 1020.

with increase in message size for both the laptop and desktop computer. This is understandable as the TX software chain needs to process more data for higher data payload size. Although the rate of increase is different for both the computers, with the 'laptop' measurements showing a faster rate of increase compared to the 'desktop computer'. The GNU Radio TX flow graph is signal processing intensive, the higher CPU clock speed and a higher number of CPU cores help the 'desktop computer' to process faster. The RX software delay ($T_8 - T_5$) is more or less constant across the different message sizes. It is primarily because of our definition of T_8, the RX software chain needs to process the same amount of data regardless of the data payload size.

The LimeSDR loopback delay ($T_5 - T_4$) decreases with increase in message size. We hypothesize this pattern is caused by the buffering of data between the LimeSDR FPGA and Cypress EZ-USB FX3 on the TX Path of the LimeSDR. The shift clock for the LimeSDR FIFOs is dependent on the sampling rate, which is 4 MHz while the buffer write clock is 100 MHz. For larger message sizes, the buffer is filled up faster at the higher clock rate. Hence the data need to shift through the buffers for significantly shorter periods of time before being transmitted using the TX signal processing chain of the LimeSDR. The measurements across both the computers have similar LimeSDR loopback delays,

TABLE II
BREAKDOWN OF THE SOFTWARE DELAYS

Message Size	GNU Radio Processing (μs)		LimeSDR Driver (μs)		User Space to Kernel Space (μs)	
	TX	RX	TX	RX	TX	RX
1	318 ± 87	325 ± 259	24 ± 43	18 ± 5	18 ± 24	10 ± 4
56	426 ± 111	315 ± 253	44 ± 46	17 ± 4	9 ± 13	10 ± 4
112	532 ± 237	295 ± 252	59 ± 82	17 ± 4	9 ± 9	11 ± 15

indicating that the placement for our time probes are correct and the measurements are independent of the host computer processing resources.

These subsequent breakdowns of the TX and RX software delays for the 'desktop computer' are shown in Table II. The GNU Radio Processing for both the TX and RX paths adds the maximum delay and jitter, with the LimeSDR driver and User Space to Kernel Space only adding very small fractions. From these results, we can highlight LimeSDR loopback delay and GNU Radio processing as our main two bottlenecks in this architecture.

Experiment 2:

Motivation It is necessary to understand how the amount of data in one bus transfer affects the overall latency (T_8 - T_1) and the LimeSDR loopback delay as it has been highlighted in previous studies [12], [14] that bus transfer delays are the most significant. However, the study of the impact of bus transfer size on the latency is missing in these previous studies.

Setup In this experiment, we vary the USB bus transfer size by varying the number of samples batched together in one transfer. We choose three different input configuration: 1020 (minimum possible by software configuration), 4080 (the default configuration) and 8160 samples in one USB transfer. The MAC data payload size is set to 56 bytes and the experiment is conducted on the 'desktop computer'.

Results Fig. 6 shows the latency, summation of the component delays, increases with increase in the number of samples. The increase of LimeSDR loopback delay contributes significantly to the increase of overall latency. It takes 255 μs to generate 1020 samples with 4 MHz sampling rate, while it takes 1020 μs and 2040 μs to generate 4080 and 8160 samples respectively. For this reason, the LimeSDR loopback delay increases with the number of samples in one USB transfer because of the queuing time increases with the increase of number of samples in one USB transfer. The RX software chain delays increases with the USB transfer size, as the GNU Radio blocks are now scheduled to process more data samples in one execution.

C. Improving the performance

The results in our previous experiments show that the LimeSDR loopback delay and GNU Radio processing delay are the two most significant delays in our system. Although the GNU Radio processing delay is quite significant, the difference in results across the two host computers indicates that these delays are because of limited host computer processing capability and can be mitigated to some extent by

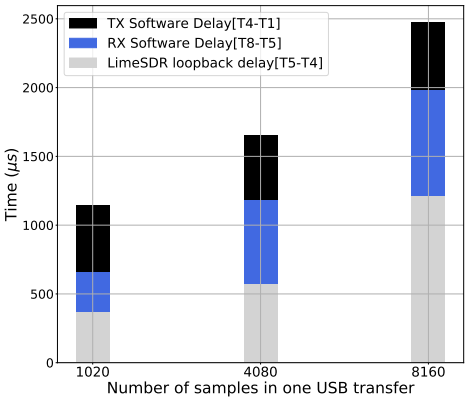

Fig. 6. **Component Delays vs Number of samples in one USB transfer**: The number of samples in one USB packet is shown along the horizontal axis, with time along the vertical axis. The component delays have been labeled with different colors. We observe the LimeSDR loopback delay and overall latency increases with the increase of number of samples in one USB Transfer.

upgrading the host computer resources. On the other hand, the LimeSDR loopback delay is more or less constant across both the computers for similar input parameters and hence can be classified as a delay contributed by the LimeSDR platform and will affect all systems using this platform. We therefore aim at decreasing the LimeSDR loopback delay.

Fig. 6 indicates that we can reduce the LimeSDR loopback delay by lowering the number of samples in one USB Transfer. The data communication between the LimeSDR and the host computer takes place in the form of LimeSDR FPGA packets. The packet size is configured in hardware to be 4096 bytes. We reconfigure the LimeSDR FPGA to use smaller packet sizes for smaller USB Transfer sizes.

In order to understand the implications of this modification we segment the RX delays into two parts: processing delay (RX software processing delay) and buffering delays (LimeSDR loopback delay). The results of Experiment 1 (Fig. 5) show that the processing delay is lower than the buffering delay. This causes inefficient utilization of the computing resources on the host computer as the processing is halted because of unavailability of samples for processing. Ideally, we want the processing delay to be equal to the buffering delay for the efficient pipelined processing of samples. Decreasing the USB transfer size helps us decrease the buffering delay but the processing overhead is increased with extra context switches, GNU Radio control signals and system calls. To obtain the best performance, we need to find the balance point of decrease of buffering delay and increase of processing overhead.

Experiment 3:

Motivation The computing resources available on the host computer dictate the processing delay and hence the balance point for a particular host computer. We need to investigate the impact of the LimeSDR FPGA packet size and host computer processing resources to find the balance point for a host computer.

Setup We use `pidstat` to monitor the resource utilization of the process on the host computer. The LimeSDR hardware is modified to use 1024 bytes, 2048 bytes and 3072 bytes as the LimeSDR FPGA packet size.

Results The result for the 'desktop computer' shown in Fig. 7

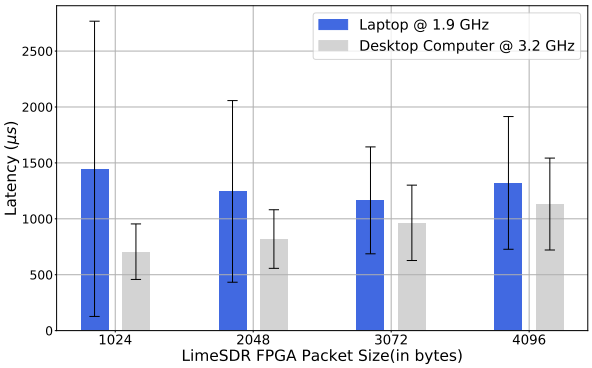

Fig. 7. **Latency (T_8 -T_1) vs LimeSDR FPGA packet size**: The LimeSDR FPGA packet size is shown along the horizontal axis and latency in μs is shown along the vertical axis. The MAC data payload size is set to 10 bytes for these results. The 'desktop computer' shows the lowest latency for the LimeSDR FPGA packet size of 1024 bytes whereas the 'laptop' has the best results for LimeSDR FPGA packet size of 3072 bytes.

exhibits that lower LimeSDR FPGA packet sizes lead to lower latency with less standard deviation. We achieve the lowest mean latency (706 μs) and standard deviation (248 μs) when we set the LimeSDR FPGA packet size of 1024 bytes as compared to 1135 μs and 414 μs for the default configuration.

On the other hand for the laptop, we obtain the lowest mean (1165 μs) and standard deviation of latency (478 μs) for LimeSDR FPGA packet size set to 3072 bytes. The results for LimeSDR FPGA sizes of 1024 bytes and 2048 bytes show high mean and standard deviations in latency which are in contrast to the trend shown in Fig. 6.

Fig. 8 shows that the 'laptop' has very high CPU usage for LimeSDR FPGA packet size of 1024 and 2048 bytes. The presence of processing overhead for smaller LimeSDR FPGA packet sizes causes the laptop processing resources to throttle as it is already processing close to its maximum capacity (95%). This lack of further processing resources results in increased buffering and unpredictable processing which explains the higher mean and standard deviation of the latency for the LimeSDR packet size of 1024 byte and 2048 bytes in Fig. 7. In case of the desktop computer, it operates at close to 87% for LimeSDR packet size of 1024 bytes, so it still has processing resources available. Hence it can continue processing the data packets at the incoming data rate resulting in lower mean and standard deviation of the latency.

These results show that we are able to decrease the latency and standard deviation in our system by decreasing the buffering delay. But the performance improvement is heavily dependent on the available host computer processing resources. The 'desktop computer' is able to achieve an improvement of 37% in mean latency with the standard deviation decreasing by

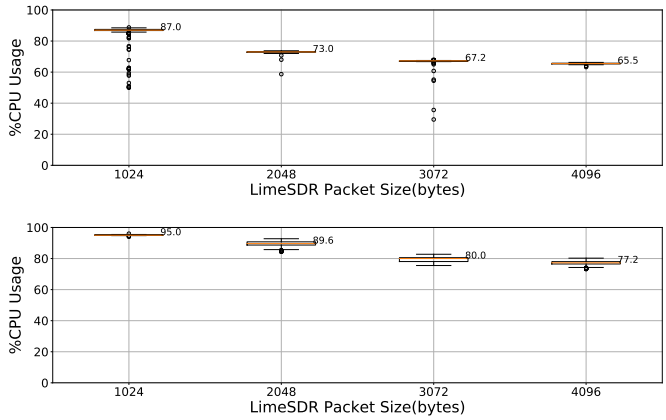

Fig. 8. **CPU utilization vs LimeSDR FPGA packet size**: The LimeSDR FPGA packet size is shown along the horizontal axis and CPU utilization is shown along the vertical axis. The results for the 'desktop computer' is shown on the top subplot with the 'laptop' shown at the bottom subplot. The MAC data payload size is set to 10 bytes for these results. Both the host computers show a trend of increase in CPU utilization with smaller LimeSDR FPGA packet size.

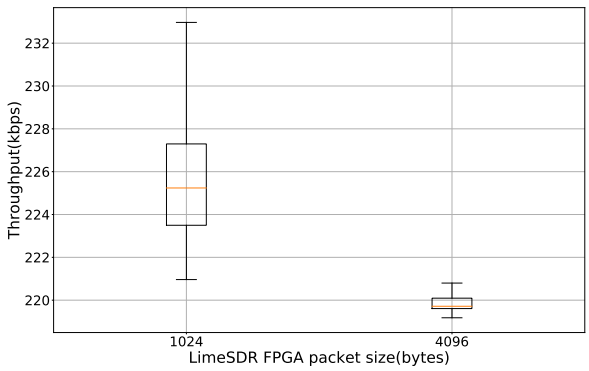

Fig. 9. **Throughput vs LimeSDR FPGA packet size**: The LimeSDR FPGA packet size is shown along the horizontal axis with throughput in kbps is shown along the vertical axis. We observe a slight increase in the throughput for LimeSDR FPGA packet size of 1024 bytes.

40%, while it was 12% and 20% respectively for the 'laptop'.

Experiment 4:

Motivation Decreasing the LimeSDR FPGA packet size decreases the buffering delays. In this case, although we achieve lower latency, the throughput of the entire system can be affected because of the increase of the processing and transfer overhead for handling the same amount of data. We need to evaluate how the throughput is affected by this modification.

Setup We send a controlled MAC data payload size from the periodic message source using the experimental setup shown in Fig. 4. The time when the SFD is detected is noted as T_{SFD}, when the complete packet is decoded, we note the time as $T_{Complete}$. We can then define throughput as:

$$\text{Throughput} = \frac{\text{MAC data payload size}}{T_{Complete} - T_{SFD}} \tag{1}$$

We compare the throughput for LimeSDR packet sizes of 1024 bytes (lowest latency) with the one for the 4096 bytes (Default configuration) for MAC data payload size of 112 bytes.

Results The results shown in Fig. 9 highlight that although the variation in throughput is much larger for the best latency case, all the measured throughput values are higher than the one for the default configuration. The median of the throughput measurements shows a slight increase of approximately 5 kbps for LimeSDR FPGA packet size of 1024 bytes. We conclude that the throughput of the system is not affected by our method to reduce delays.

D. Discussion

Even with these improvements, we cannot match the timing specification required for the round-trip acknowledgment time of IEEE 802.15.4 with a software-only implementation. Hence, we need to relax the timing specifications in order to support IEEE 802.15.4 devices using this architecture. Other protocols like BLE and LoRa that have less strict timing requirements [20], [21] are more suitable for our architecture. Another possibility could be to implement Time Division Multiple Access (TDMA) protocols such as Time Slotted Channel Hopping (TSCH) [22] using our architecture. TSCH lets us define our own timeslot timing template and the round trip acknowledgment time can also be defined. For TDMA protocols, we will need to address the problems of timing synchronization with other nodes in the presence of these delays.

V. Related Work

In this section, we discuss the previous relevant studies in two different contexts: studies related to architectures for heterogeneous IoT gateways and previous work on the delay analysis in SDRs.

A. IoT gateway architecture context

Gioia et al. [8] discuss the development of the AMBER gateway based on three main tenets: scalability, flexibility and modularity. The modularity is provided by 'Extender' sockets which connect to dedicated transceivers for the the supported communication protocols. Previous studies [7], [9], [11] propose similar gateways with added features like security and capillary networks.

Morabito et al. [10] and Karhula et al. [23] propose an edge-based architecture for the IoT gateway. The different functionality of the gateway is virtualized using containers similar to VGATE. But these architectures use regular hardware modules instead of SDRs thus limiting the scope of supported protocols and experimentation offered by VGATE.

Surligas et al. [24] show the possibility of simultaneous operation of IEEE 802.11 and IEEE 802.15.4 using a SDR. In their work, they design dedicated kernel drivers for each of these protocols. Dongare et al. [25] propose a cloud-based LoRaWAN gateway using SDR. Their work highlights the that coherent combination of received samples across multiple

gateways can be used to increase the range and battery life of the sensor nodes. These works highlight the flexibility of an SDR based gateway can be leveraged for providing better connectivity to multiple protocols.

Narayanan et al. [26] propose an SDR-based IoT gateway to address the problem of cross-technology interference. In their architecture, they use the cloud to concurrently decode LoRa, Zigbee and Z-Wave using the physical layer differences across the different protocols. One of the central assumptions in this work is that these protocols have lax latency requirements. In contrast, we consider latency as an important metric, and hence we highlight and minimize the problems associated with latency in our work.

B. SDR delay analysis context

A number of previous research articles [12], [13], [14] study the delays inherent in an USRP platform based IEEE 802.15.4 setup. These previous works focus on coarse-grained measurements whereas in this study we focus on comprehensive evaluation with timestamps at each individual component. In the VGATE architecture, we envision a software transceiver and hence took a system evaluation approach as compared to measuring these delays in isolation. This approach takes into consideration the impact of the system computational load on the delays, which is missing in these previous works.

Truong et al. [13] and Puschmann et al. [27] have showcased that the overall latency in USRP based setups can be decreased by USRP driver buffer tuning. We improve upon their work by addressing the queuing delay in the SDR buffers.

VI. Conclusions

This paper introduces the VGATE architecture for heterogeneous IoT gateways. In this paper, we study the delays associated with the use of SDRs and highlight the GNU Radio processing time and USB bus transfer time as the major delays. We investigate the impact of USB Transfer size and host computer resources and reduce the mean and standard deviation of latency by 37% and 40% respectively.

Acknowledgments

This work has been partially funded by the H2020 collaborative Europe/Taiwan research project 5G-CORAL (grant num. 761586).

References

[1] Ericsson, "Internet of things outlook." online, 2017.

[2] C. Doukas and I. Maglogiannis, "Bringing iot and cloud computing towards pervasive healthcare," in *International Conference on Innovative Mobile and Internet Services in Ubiquitous Computing*, pp. 922–926, IEEE, 2012.

[3] A. Zanella, N. Bui, A. Castellani, L. Vangelista, and M. Zorzi, "Internet of things for smart cities," *IEEE Internet of Things journal*, vol. 1, no. 1, pp. 22–32, 2014.

[4] J. Lee, E. Lapira, B. Bagheri, and H.-a. Kao, "Recent advances and trends in predictive manufacturing systems in big data environment," *Manufacturing Letters*, vol. 1, no. 1, pp. 38–41, 2013.

[5] M. T. Lazarescu, "Design of a wsn platform for long-term environmental monitoring for iot applications," *IEEE Journal on emerging and selected topics in circuits and systems*, vol. 3, no. 1, pp. 45–54, 2013.

[6] H. Derhamy, J. Eliasson, J. Delsing, and P. Priller, "A survey of commercial frameworks for the internet of things," in *IEEE International Conference on Emerging Technologies and Factory Automation*, IEEE Communications Society, 2015.

[7] A. Amiruddin, A. A. P. Ratna, R. Harwahyu, and R. F. Sari, "Secure multi-protocol gateway for internet of things," in *Wireless Telecommunications Symposium*, pp. 1–8, IEEE, 2018.

[8] E. Gioia, P. Passaro, and M. Petracca, "Amber: An advanced gateway solution to support heterogeneous iot technologies," in *Software, Telecommunications and Computer Networks*, pp. 1–5, IEEE, 2016.

[9] N. Gyory and M. Chuah, "Iotone: Integrated platform for heterogeneous iot devices," in *International Conference on Computing, Networking and Communications*, pp. 783–787, IEEE, 2017.

[10] R. Morabito, R. Petrolo, V. Loscri, and N. Mitton, "Legiot: a lightweight edge gateway for the internet of things," *Future Generation Computer Systems*, vol. 81, pp. 1–15, 2018.

[11] J. Kaur and M. Singh, "Multiprotocol gateway for wireless communication in embedded systems," *International Journal of Computer Applications*, vol. 72, no. 18, 2013.

[12] T. Schmid, O. Sekkat, and M. B. Srivastava, "An experimental study of network performance impact of increased latency in software defined radios," p. 59, ACM Press, 2007.

[13] N. B. Truong and C. Yu, "Investigating Latency in GNU Software Radio with USRP Embedded Series SDR Platform," in *International Conference on Broadband and Wireless Computing, Communication and Applications*, pp. 9–14, Oct. 2013.

[14] G. Nychis and T. Hottelier, "Enabling MAC Protocol Implementations on Software-Defined Radios," in *USENIX Symposium on Networked Systems Design and Implementation*, pp. 91–105, 2009.

[15] M. Satyanarayanan, "The emergence of edge computing," *Computer*, vol. 50, no. 1, pp. 30–39, 2017.

[16] C.-Y. Chang, N. Nikaein, R. Knopp, T. Spyropoulos, and S. S. Kumar, "Flexcran: A flexible functional split framework over ethernet fronthaul in cloud-ran," in *IEEE International Conference on Communications*, pp. 1–7, IEEE, 2017.

[17] B. Montazeri, Y. Li, M. Alizadeh, and J. Ousterhout, "Homa: A receiver-driven low-latency transport protocol using network priorities," *arXiv preprint arXiv:1803.09615*, 2018.

[18] M. Miao, F. Ren, X. Luo, J. Xie, Q. Meng, and W. Cheng, "Softrdma: Rekindling high performance software rdma over commodity ethernet," in *Proceedings of the First Asia-Pacific Workshop on Networking*, pp. 43–49, ACM, 2017.

[19] B. Bloessl, C. Leitner, F. Dressler, and C. Sommer, "A GNU Radio-based IEEE 802.15.4 Testbed," in *12. GI/ITG KuVS Fachgespräch Drahtlose Sensornetze*, pp. 37–40, September 2013.

[20] J. Nieminen, C. Gomez, M. Isomaki, T. Savolainen, B. Patil, Z. Shelby, M. Xi, and J. Oller, "Networking solutions for connecting bluetooth low energy enabled machines to the internet of things," *IEEE network*, vol. 28, no. 6, pp. 83–90, 2014.

[21] LoRa Alliance, *LoRaWAN Specification*, 1 2015. V 1.0.

[22] "IEEE Standard for Low-Rate Wireless Networks," *IEEE Std 802.15.4-2015 (Revision of IEEE Std 802.15.4-2011)*, pp. 1–709, Apr. 2016.

[23] P. Karhula, J. Mäkelä, H. Rivas, and M. Valta, "Internet of things connectivity with gateway functionality virtualization," in *Global Internet of Things Summit*, pp. 1–6, IEEE, 2017.

[24] M. Surligas, A. Makrogiannakis, and S. Papadakis, "Empowering the iot heterogeneous wireless networking with software defined radio," in *Vehicular Technology Conference*, pp. 1–5, IEEE, 2015.

[25] A. Dongare, R. Narayanan, A. Gadre, A. Luong, A. Balanuta, S. Kumar, B. Iannucci, and A. Rowe, "Charm: exploiting geographical diversity through coherent combining in low-power wide-area networks," in *International Conference on Information Processing in Sensor Networks (IPSN)*, pp. 60–71, IEEE, 2018.

[26] R. Narayanan and S. Kumar, "Revisiting software defined radios in the iot era," in *Proceedings of the 17th ACM Workshop on Hot Topics in Networks*, HotNets '18, pp. 43–49, ACM, 2018.

[27] A. Puschmann, M. A. Kalil, and A. Mitschele-Thiel, "Implementation and evaluation of a practical SDR testbed," in *Proceedings of the 4th International Conference on Cognitive Radio and Advanced Spectrum Management*, pp. 1–5, ACM Press, 2011.