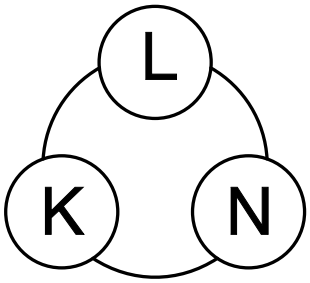


**Technische Universität München**
**Lehrstuhl für Kommunikationsnetze**
Prof. Dr.-Ing. Wolfgang Kellerer

# Research internship report

Eastbound interface development and latency evaluation for industrial wireless testbed.


| | |
|---|---|
| Author: | Rajathadripura Kumaraiah, Yadhunandana |
| Address: | Schröfelhofstraße 14 |
| | 81375 Munich |
| | Germany |
| Matriculation Number: | 03680943 |
| Supervisor: | Gürsu Murat, Samuele Zoppi |
| Begin: | 07. August 2017 |
| End: | 29. September 2017 |

With my signature below, I assert that the work in this thesis has been composed by myself independently and no source materials or aids other than those mentioned in the thesis have been used.

| München, 11.06.2014 | |
|---|---|
| Place, Date | Signature |



| München, 11.06.2014 | |
|---|---|
| Place, Date | Signature |

# Abstract

The eastbound interface in WSN (wireless sensor network) is the communication interface between host computer. In our experimental setup, the host computer is running an industrial application, it communicates with the mote using serial interface. This interface is considered as eastbound bound interface.

This document describes the development and latency characterization of eastbound interface. The purpose of characterization is to evaluate latency in capillary networks. Wireless communication has become integral part of our lives however still it has not succeeded in finding widespread use in industrial or automation field. It is mainly because of the stringent reliability and latency requirements. Recent technological innovations in wireless communication field can achieve these stringent requirements.

In this work, eastbound interface is developed to evaluate the components which cause the latency, both from hardware as well as from software perspective. Presently many wireless sensor network platforms and operating systems are available, most of them are COTS (commercial of the shelf)solutions. To the best of our knowledge we did not find any platform best suited for evaluating stringent delay and reliability requirements of industrial communication. In this work, an existing hardware platform Z1 from Zolertia$^{TM}$and open-source protocol stack OpenWSN is considered as a starting point for the development of testbed. An extendable eastbound interface is developed for controlling/injecting data to mote. This facilitates communication with other nodes in the network. Delay contribution due this interface is analyzed both analytically and measured practically. Additionally MAC layer is modified to support shorter slots (from 15ms to 6ms) and to better support mote communication with host computer.

In addition to development of mote firmware, a python framework is developed which runs in the host computer and performs following functions, controlling the mote, maintaining the routing table in case the mote is network coordinator, collects the network statistics when needed and connects to the application running in the host via UDP sockets.The data read via UDP sockets is injected to mote for communicating with the other end of application.

Finally extensive analysis of eastbound interface and communication stack processing delay are presented.

# Contents

# Chapter 1

# Introduction

Protocols such as Ethernet, Foundation Fieldbus, Profibus, and HART are well established in the industrial process control space. Until now industrial communication was mainly through wired networks such as Ethernet, Modbus etc. Due to their reliability, latency and robustness in industrial environment.

Recent innovations in wireless technologies and hardware have made RF chips cheap, reliable and robust. IC Insights predict that the average selling price (ASP) of mobile-device analog ICs, in other words wireless IC's will decline by more than 30% from 2011 to 2019[1]. This opens new era in industrial communication, where machines, manufacturing processes can be monitored remotely without tedious and costly wired networks. Wireless nodes being easily installable and replaceable become an attractive option. With all these advantages, there are problems which needs to be solved. For example, to satisfy stringent latency and reliability requirements, new type of medium access methodologies have to be developed, implementations need to be optimized for achieving low latency and to maintain network reliability, new routing algorithms have to be developed. These new implementations have to consider these stringent requirements while defining the objective function. Stake in manufacturing environments is very high. A network disconnection might result in a losses in the order of billions.

In this internship, development of eastbound interface and its characterization is carried out for industrial wireless sensor network testbed. Testbed is then used for evaluating wireless sensor networks for latency and reliability. To develop such a testbed OpenWSN software stack is considered. It provides an open source implementation of IEEE802.15.4e "Time Synchronized Channel Hopping" medium access standard, which achieves high reliability through frequency agility (channel hopping). This medium access mechanism achieves both reliablity and low power in unreliable wireless channel through time synchronization with other nodes. The development of platform is done on the commercially available of the shelf hardware platform Zolertia Z1. The main focus of this research internship was

[1] http://www.icinsights.com/news/bulletins/IC-Insights-Raises-2017-IC-Market-Forecast-To-22-/

development of a testbed for analyzing delay contributors, low latency network setup and latency characterization of eastbound interface of the network.

In Chapter 2, brief introduction of OpenWSN stack, TSCH and LLDN are given. OpenWSN, an implementation of protocol stack that we use throughout this work, is introduced in this chapter. We briefly explain TSCH and LLDN mechanism.

In Chapter 3, details of our implementation are presented. The first part explains the design of the openserial driver module and minimal network management python module. The second part describes the LLDN schedule construction. The third part briefly describes modification of external MAC. The fourth part presents extensive characterization of eastbound interface. In the fifth part communication stack processing delay for OpenWSN stack is presented. At last, different delay contributors in the wireless sensor network are compared with eastbound interface delay.

In Chapter 4, conclusion to the work is given and a glimpse of future research is offered.

# Chapter 2

# Background

Industrial domain has opted wireless sensor networks over traditional wired communication due to several attractive features of IWSNs such as self-organization, rapid deployment, flexibility. Explosive growth has occurred in the use of wireless sensor networks in a variety of applications, since wireless technology can reduce costs, increase productivity, and ease maintenance. However the advantages come with some challenges which needs to be addressed.

The major technical challenges involved for the deployment of IWSNs are outlined as follows.

**Resource constraints:** The wireless nodes are constrained in terms of memory, power and the processing power.

**Dynamic topologies:** Routing between the central hub and leaf nodes is done through hop by hop, so if one node fails in the path, network should be able to find another route to communicate with central node (Not applicable for single hop).

**Latency requirements:** The wide variety of use cases conceived as possibility on IWSNs will have different QoS requirements and specifications. The QoS provided by IWSNs refers to the accuracy between the data reported central hub (industrial controller) and what is actually happening in the plant. In addition, since sensor data are typically time-sensitive, e.g., receiving the sensor readings on time in a control loop, it is important to receive the data at the sink in a timely manner. Data with a long latency due to processing or communication may be outdated and lead to wrong decisions in the industrial controller.

**Packet errors and variable-link capacity:** Compared to wired networks, in IWSNs, the attainable capacity of each wireless link depends on the interference level experienced at the receiver, and high bit error rates (BER=$10^{-2}$-$10^{-6}$) are observed in communication. In addition, wireless links exhibit widely varying characteristics over time and space due to obstructions and noisy environment. Thus, capacity and delay attainable at each link are location-dependent and varies continuously, making QoS provisioning a challenging task.

**Integration with Internet and other networks:** It is of fundamental importance for the commercial development of IWSNs to provide services that allow querying of the network to retrieve useful information from anywhere and at any time. For this reason, the IWSNs should be remotely accessible from the Internet therefore IWSNs need to be integrated with the Internet Protocol (IP) architecture.

## 2.1 OpenWSN Stack

OpenWSN is an open source internet of things stack based on highly reliable TSCH based the the MAC layer protocol 802.15.4e. The stack implements RPL procotol for upward routing and source routing for downward traffic. The stack uses 6LowPAN. 6LowPAN is an IPv6 over low power wireless personal area networks, it defines the encapsulation and header compression mechanism for sending and receiving IPv6 packets over 802.15.4 networks.

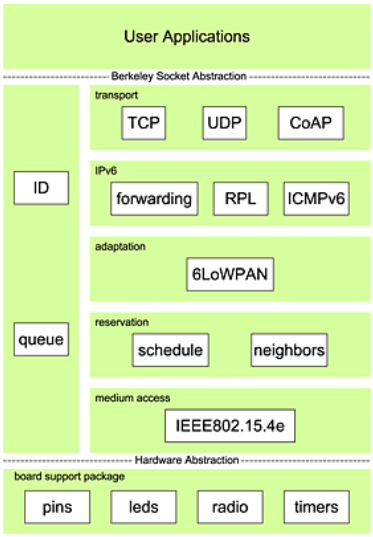


Figure 2.1: OpenWSN stack[WVK+12]

## 2.2 TSCH

Reliability and deterministic behavior is must in industrial communication with low power, since nodes are running with scavenged or battery power. TSCH is medium access mechanism inspired from WirelessHART and ISA100.11a. TSCH provides both low power and high reliability. In TSCH mechanism nodes in the network are time synchronized with the super frame and cells (pair of RX and TX slots between nodes) are allocated between the nodes. Nodes have to turn the radio on only during their slot otherwise radio will be

turned off. This method in addition to saving power reduces packet collisions, since only one node can transmit packet in a particular slot. 6tisch is an implementation of TSCH over IPv6.

## 2.3 LLDN

Factory automation applications require large number of nodes to observe and control production. Nodes need to collect data from robots, portable machine tools, such as milling machines. This applications require high determinism, reliability and low latency. LLDN (Low Latency Deterministic Network) standard of IEEE considers the latency as main QoS element.

LLDN supports only star topology due to low latency requirements, LLDN superframe is a time division multiple access (TDMA) scheme. In LLDN nodes are synchronized only with beacons frames. Sychronization with acks is not possible, since acks are removed to reduce latency.

## 2.4 Problem description

OpenWSN is an open source implementation of a 6tisch standard coupled with internet of things standards such as 6LoWPAN, RPL, CoAP. The OpenWSN stack allows low power ultra reliable communication, However design is not optimal for low latency communication. Out of the box it is very difficult to use OpenWSN for evaluating capillary networks for latency and reliability. In this research internship, the task is to improve the existing OpenWSN stack to use it for evaluating latency bottlenecks and also to set up low latency deterministic network. In the internship new modules, APIs are added and existing implementations are improved for achieving the desired goal.

# Chapter 3

# Implementation and Results

## 3.1 Development and evaluation eastbound interface

Wireless sensor network testbeds allow researchers to conduct experiments, evaluate different channel access algorithms and protocols. In industrial communication latency and reliability play significant role, therefore in our implementation we are concentrating more on these parameters. With a wireless testbed it is easy conduct experiments and evaluate different algorithms.

In the process of developing a low latency wireless testbed, the first step is setting up low latency network in which latencies can be measured or characterized accurately. The road block for this was OpenVisualizer. OpenVisualizer is necessary for setting WSN. however OpenWSN was making latency measurement/debugging difficult. Since it was sending huge amount of debugging information to host through eastbound interface. This data is not essential for network functionality. Stack data is sent to host continuously to make system user friendly. Eastbound traffic was not user configurable and it was increasing latency of the overall system. The next step was to figure out a way to setup WSN network without OpenVisualizer, which involved understanding all the functionalities OpenVisualizer is doing and identifying which functionalities are absolutely necessary.

Through thorough investigation of OpenVisualizer code, following functionalities are identified.

1. Maintaining routing table for nodes in the network for downward traffic using source routing headers.
2. Communicating with motes at the exact serial RX and TX frame slots of schedule.
3. Controlling/configuring the parameters of sensor network.
4. Packets read from TUN/TAP interface, converted to 6LoWPAN format with source

routing header and then injected to mote via serial interface.

5. Lot of unwanted debug information is sent from stack to OpenVisualizer to make it user-friendly, for which at least 2 slots (each slot 15 milliseconds) are used, hence making super frame long and resulting in more latency.
6. Modules for redirecting packets received from the sensor network to wireshark for debugging.
7. HDLC protocol implementation.

All this data is not sent to the host when the host is requesting for it. This is lot of traffic, which causes significant latency in overall system.

In the above findings, functionality which contributed most to latency and made latency characterization difficult is the 5, Lot of unnecessary debug information was being sent and it was not critical. The next component was TUN/TAP interface which is not needed unless we want to put the data to internet therefor it was making the OpenVisualizer heavy and complex.

The next step for reducing latency and making measurements easy is to kill OpenVisualizer then develop host side tool replacing OpenVisualizer which has minimal functionality necessary for maintaining the network. This host side tool is developed in Python. In addition to maintaining network, it should ease the characterization of latency in different components of the system.

OpenVisualizer communicates with openserial driver running in OpenWSN firmware. openserial driver needs to be redeveloped to comply with the new functional behavior of Python module (instead of OpenVisualizer) running in the host. Here new serial packet formats are developed to modularize processing of the control, data, debug and error frames.

### 3.1.1 Openserial driver design

In the original implementation, data is exchanged over hdlc protocol. This protocol further introduces overhead over serial communication. This hdlc protocol is removed to avoid overhead.

Another problem with openserial driver stack is that information is sent to OpenVisualizer without being asked for. In our design openserial driver sends the necessary data only when it is requested due to this design decision, only required data when it is requested is sent to host. In this design 15ms slots are no longer necessary so the slot size can be reduced.

The scheduling of the serial driver is carried out according to super frame timing. openserial driver APIs `openserial_startOutput`,`openserial_startInput`,`openserial_stop` are the

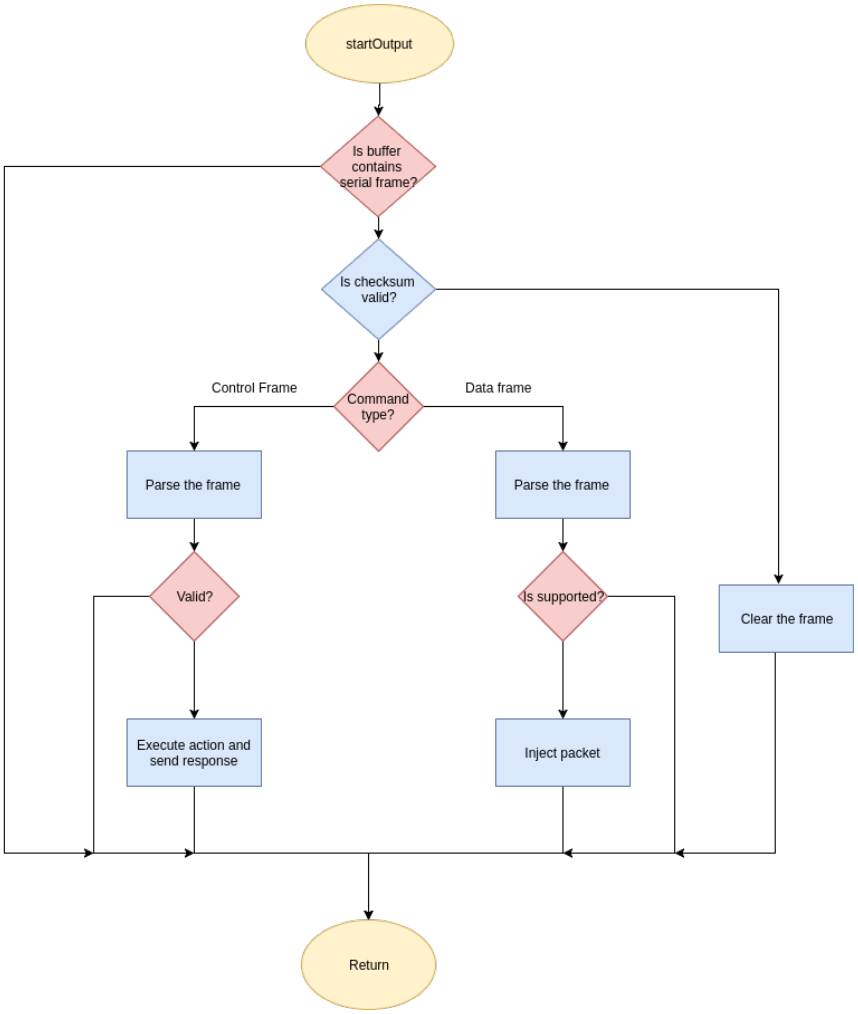


Figure 3.1: Serial frame processing

function which do all the work.

`openserial_startOutput` reads the data queued in the output circular buffer, transmits this data to host via uart in the SERIALTX slot.

`openserial_stop` processes the serial data received from the host in SERIALTX slot, before starting the uart transmission.

`openserial_startInput` sends the request frame to host for indicating the SERIALRX slot so that host can send the data.

Above three APIs were implemented in OpenWSN originally to schedule the openserial component from MAC layer. In the present design also scheduling is kept same as the original one. Except modification of MAC layer related to serial communication as explained in the subsection 3.3.

In the present implementation of openserial driver, buffer management and serial data parsing are the two main components. Lets have a look briefly into these components.

**Buffer management**:
Open serial driver manages the data in two circular buffer implementations TX_BUFFER and RX_BUFFER each of size 256 bytes. When SERIAL_RX slots are scheduled, host sends data via uart. Once the byte is received in the interrupt handlers, data is pushed in to SERIAL_RX buffer. This buffered data is processed in the idle slot if any, before mote goes in to sleep mode or in SERIAL_TX slot by calling `openserial_stop` at the beginning of SERIAL_TX slot.

TX_BUFFER is used for buffering the data, which needs to be sent via serial to host computer. Whenever mote wants send data (eg: sending packet received or debug information), it calls `openserial_printf`. This function is part of openserial driver which buffers data to TX_BUFFER. When SERIAL_TX slot is scheduled, this data is read from TX_BUFFER and transmitted via UART. As will be explained later in this chapter, With the aim of reducing latency, slot widths in TSCH schedule are reduced as minimal as possible. During this research internship slot widths are reduced from 15ms to 6ms, so only a limited amount of data can be sent via serial. One can calculate how much data can be sent via eastbound interface at a particular baudrate with the following equation.

$$Max\_bytes\_per\_slot = (\frac{baudrate}{10}) * slot\_width \tag{3.1}$$

For example at a baudrate of 115200 and 6ms slot width, maximum 69 bytes can be sent. These numbers have to be kept in mind while sending data via serial. Here while calculating the slot width additional margin of 0.6 ms for USB controller stack and chip processing delay must be taken into account.

In the openserial driver to make implementation modular, two types of serial frames are implemented. Data and command serial frames, which are used for packet injection and network management respectively. For example, serial frames used for injecting UDP packet belong to serial data frames category, serial frames used for receiving TSCH schedule belong to command serial frame category. This type modularization helps to modularize the serial frame processing logic in openserial driver.

To make serial frames processing streamlined, every serial packet starts with `0x7e` which indicates beginning of the frame, next byte represents overall length of the packet then comes packet type, subtype and payload. Lets look at sample serial frames.

Following command frame configures the mote as DAGroot. The first byte indicates beginning of the frame, The second byte length of the packet excluding `0x7e`. The third byte indicates that it is a control frame, fourth byte represents action of the command, i.e to set mote as dagroot.

| 0x7e | 0x03 | 0x43 | 0x00 |
|---|---|---|---|

Table 3.1: Serial frame set DAG root

Following sample command represents command used for injecting UDP packet to network, This frame also follows the same protocol explained above. 3rd byte represents that command type is data, fourth byte represents the specific command type such as UDP injection or TCP injection.

| 0x7e | 0x0b | 0x44 | 0x00 | 0xff | 0x02 | 0x02 | 0x02 | 0x02 | 0x02 | 0x58 | 0x01 |
|---|---|---|---|---|---|---|---|---|---|---|---|

Table 3.2: Serial frame, Inject UDP packet

| | |
|---|---|
| 0x7e | Indicates the beginning of packet |
| 0x0b | Length of the packet excluding 0x7e |
| 0x44 | Indicates the category of packet('D') data packet |
| 0x00 | Subcategory of the packet UDP inject |
| ff0202020202 | UDP payload |
| 5801 | Check sum |

Table 3.3: Different fields in a packet

The command category, byte in the third position is very useful for modularizing the processing logic in the openserial driver. During packet processing based on the third byte openserial driver delegates the processing of packet to the respective module (command or data processing category).

### 3.1.2 Network management module

After openserial driver designed, host side application is needed for handling following functionalities. This a minimal python module implements only the absolutely required functionality. Design idea behind minimal network management module in contrast to OpenVisualizer is that, mote shares the network stats only when this module requests for it (few exceptions like critical errors and when packet arrived).

1. For configuring the network(Setting DAG Root).
2. Getting network statistics.
3. Measuring the latency from stack.

4. Maintaining network topology for downward source routing.
5. Setting up required TSCH schedule for nodes.
6. Forms 6LoWPAN packet from industrial control system data.
7. Handles serial communication.

Lets go into details of the each functionality, briefly look at the purpose they serve.

1. **For configuring the network(Setting DAG Root):** When all the nodes are running for the network to start functioning at least one node has to DAG root it acts like a central collection node. DODAG is formed according to APL protocol by keeping this node are root node. Network forming starts from here, i.e as soon as this node is made root node it transmits the advertisement packets about presence of the network to other nodes to join the network. Therefore one node has to be dagroot the functionality is achieved by sending a serial command frame.
2. **Getting network statistics:** Getting network statistics is very important for debugging and to get information about network such as schedule, neighbors etc. This functionality is again implemented based command-response basis. When the host wants to know stack information corresponding command is sent, network statistics are returned in the response frame.
3. **Measuring the latency from stack:** Latency plays a very significant role in industrial communication. Measuring latency functionality is required, Latency may be due to communication stack, MAC schedule or serial etc. To make these measurements easy, few options are provided in the network management module, where the stack injects data, latency info is taken from stack sent through response frame, this information is saved in JSON format for further processing.
4. **Maintaining network topology for downward source routing:** In OpenWSN stack upward routing is based on Routing Protocol for Low-Power and Lossy Networks (RPL), which is based on DODAGs as explained in [GK12]. RPL works for upward routing and it is specifically designed for data collection networks. OpenWSN achieves downward routing using source routing mechanism. Packet which needs to routed contains routing path in the header. Each node in the path looks at the packet and forwards according to the route present. DAGroot runs only the MAC layer in OpenWSN stack. When packet needs to sent downward from DAGroot, packet should contain routing information, hence DAGroot must be aware of the complete network. This is achieved in OpenWSN with DAO messages. All the nodes of the network send information about their parents to DAGroot, this information is forwarded to the host application. Routing table is built based on nodes parent information. Before injecting packet to DAGroot host side application creates source routing header. Similar implementation is adapted to network management module.
5. **Setting up required TSCH schedule for nodes:** Handling schedule is one more

feature required for managing the schedule according to traffic/latency requirements, therefore three specific commands are implemented to view schedule, add or remove RX and TX slots.

6. **Forming 6LoWPAN packet from industrial control system data:** In the host computer, industrial control system (sensor/controller) is running and it sends calculated/sensed data via sockets. This data needs to be received and converted to 6LoWPAN packet. This functionality is also implemented in network management python module.
7. **Handling serial communication:** Serial communication functionality is implemented through multiple threads avoid latency in the host application. Serial communication is handled by separate threads with necessary synchronization between them.


Figure 3.2: Experimental setup

## 3.2 LLDN schedule construction

To achieve low latency the MAC layer is modified, the 802.15.4e MAC mechanism allows synchronization with acks. However the synchronization based on acks makes slot widths long, which in turn increases latency. To avoid increased latency, acks are removed from the schedule and synchronization of nodes is done only through beacon frames with their frequency increased. Schedule is designed as explained in [Oez16]. Sample schedule is shown below.

In the schedule, the TXRX slot is used for sending beacon frames, hence allowing all nodes in the network (star topology) to synchronize with their parents. In this case with DAGroot.

Typical schedule in the present implementation.

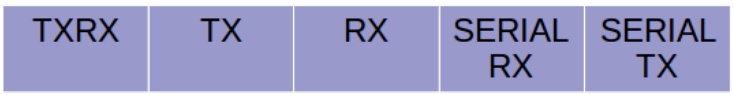


Figure 3.3: schedule

## 3.3 Modified OpenWSN External MAC

External MAC is a mechanism to decide when the data has to be sent from user application running in host to the mote and vice versa. Since the mote has single microcontroller this communication time also should be part of super frame schedule. In order to do that, serial communication timing is accommodated in the schedule through serial RX and serial TX slots. These slots are used for communicating with the host computer running user application. The mechanism is as explained below. In the schedule as soon serial RX slot starts, MAC layer calls the start input function of the openserial driver, then a request frame is sent to host to indicate that it is ready to receive. Once the host receives the request frame it sends the data to mote. In the original OpenWSN implementation, request frames were sent once in every super frame. In this design, user application cannot inject multiple data packets in one superframe although enough slots are available. This is not well suited for achieving low latency communication. In our design the MAC layer is modified in such a way that for every SERIALRX slot a request frame is sent to the host.

## 3.4 Eastbound delay characterization

This section describes the characterization of delays involved in the eastbound interface of wireless sensor network setup.

### 3.4.1 Setup

In the experimental setup Zolertia-Z1 mote running OpenWSN is considered. Mote is connected to the host running user application via serial interface. The packets generated in the application are injected into the mote through serial interface. For Z1 mote, serial communication is done over a USB to serial converter module. This serial interface involves USB to UART converter from Silicon Laboratories (CP2102). This chip converts the serial data to USB packets (conversion introduces the delay).

USB is not an ideal communication protocol for achieving real time communication, especially, if the slave device is a bulk type device. The USB controller schedules the bus access to the bulk devices based bus availability therefore latency is not guaranteed. This latency varies depending on how many devices are connected to the USB controller and also on the type of end points they have. For example, if more interrupt and isochronous end point devices are connected then bulk devices will be deprived from the bus access, since interrupt and isochronous devices have higher priority than bulk devices. When the mote sends data to host there is one more additional delay involved called USBtimeout value of the chip. This delay is caused, because the CP2012 chip waits for the next byte for (18/baudrate) duration before forming a USB packet and transmitting it. In the current

design host sends data to mote only after receiving a request frame. When SERIALRX slot starts, mote sends request frame. Due to the USBtimeout duration the request frame is delayed by USBtimeout value. This delay adds to the overall latency of the system.

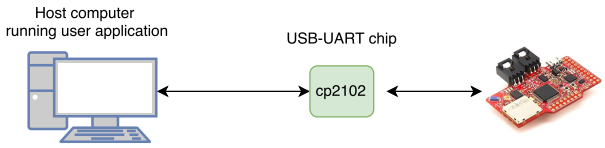


Figure 3.4: Eastbound characterization setup.

### 3.4.2 Evaluation

Latency in the eastbound interface has a contribution mainly from four components. Processing and USB polling delay at the host ($T_{host}$), USB transmission and protocol overhead delay ($T_{usb}$), UART transmission delay ($T_{serial}$) and USB-UART chip processing delay ($T_{chip}$) the total delay is sum these delay components as represented by the following equation.

$$T_{total} = T_{host} + T_{usb} + T_{chip} + T_{serial} \tag{3.2}$$

The $T_{host}$ specific delay is due to USB driver and USB controller. In the experimental set up this value varies as plotted in the figure 3.6 on page 20.

The protocol overhead delay is due to inherent nature of the USB protocol. This delay comprises of the time involved in sending IN,OUT,ACK packets plus the data transmission time. Once the data from host is received at the USB-UART converter chip, this data is transmitted to the mote through serial communication. This delay is highest among all the delays and it directly depends on the baud rate used. In the experimental setup, 115200 baud rate is used, which results in 86.6 microseconds delay for transmitting a single byte.

Following figure shows total eastbound delay versus data size. The delay is almost a linear function of data size. This dependency is driven mainly by $T_{serial}$.

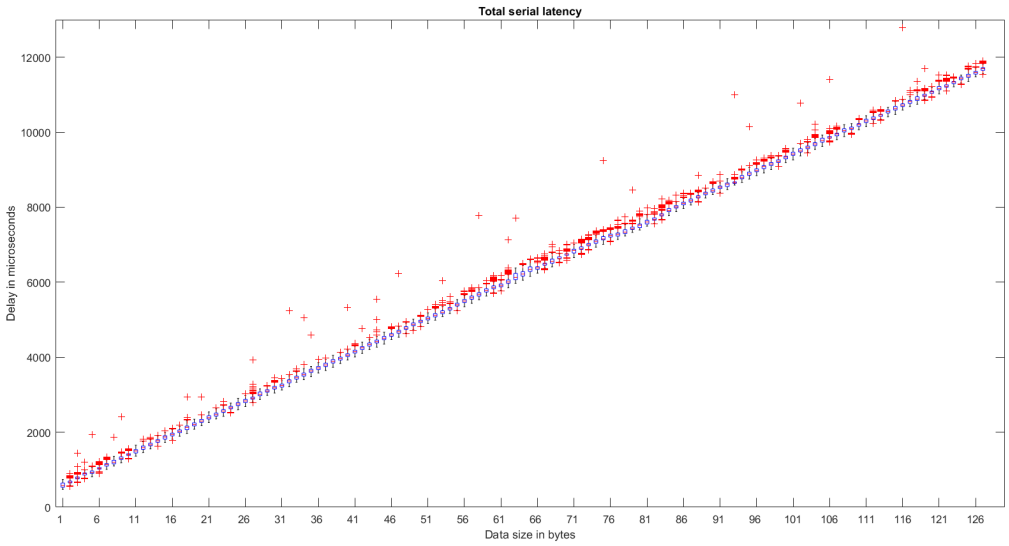


Figure 3.5: Total serial latency

At last, the chip delay. This delay is measured by taking USBmon logs to identify how much delay chip is taking for sending acks after receiving the USB packet. This delay was also found to be significant. Please see the figure 3.9 on page 22.

For the evaluation of latency, four delay contributors should be considered separately. $T_{host}$ host specific delay is obtained by subtracting chip ack delay, UART delay and USB overhead and it is obtained from the measurements.

#### 3.4.2.1 Chip ack processing delay

This delay is the time taken by the CP2102 USB-UART chip to send the ack after USB packet is received. With the measurements from USBmon logs it is found to be very significant. This values found to be stochastic except for the USB timeout value.

As discussed earlier, the CP2102 chip has a parameter called USBtimeout. This is the time after which the CP2102 chip starts sending received uart bytes as usb packets. This value is dependent on the baud rate. At 115200 baud rate the USBtimeout value is 156 $\mu$s. This delay is required when data is sent from mote to the host, Since the chip waits for 156 $\mu$s before sending data to the host. This value is constant hence not included in chip ack delay plot.

The chip ack processing delay is weakly dependent on data size. It increases slightly with data size. This increase is significant when data size is more than 64 bytes. When the packet is greater than 64 bytes, packet needs to be sent in two fragments, therefore chip

has to wait for two chunks. This increases the chip ack processing delay. Chip ack delay values are plotted in figure 3.6 on page 20.

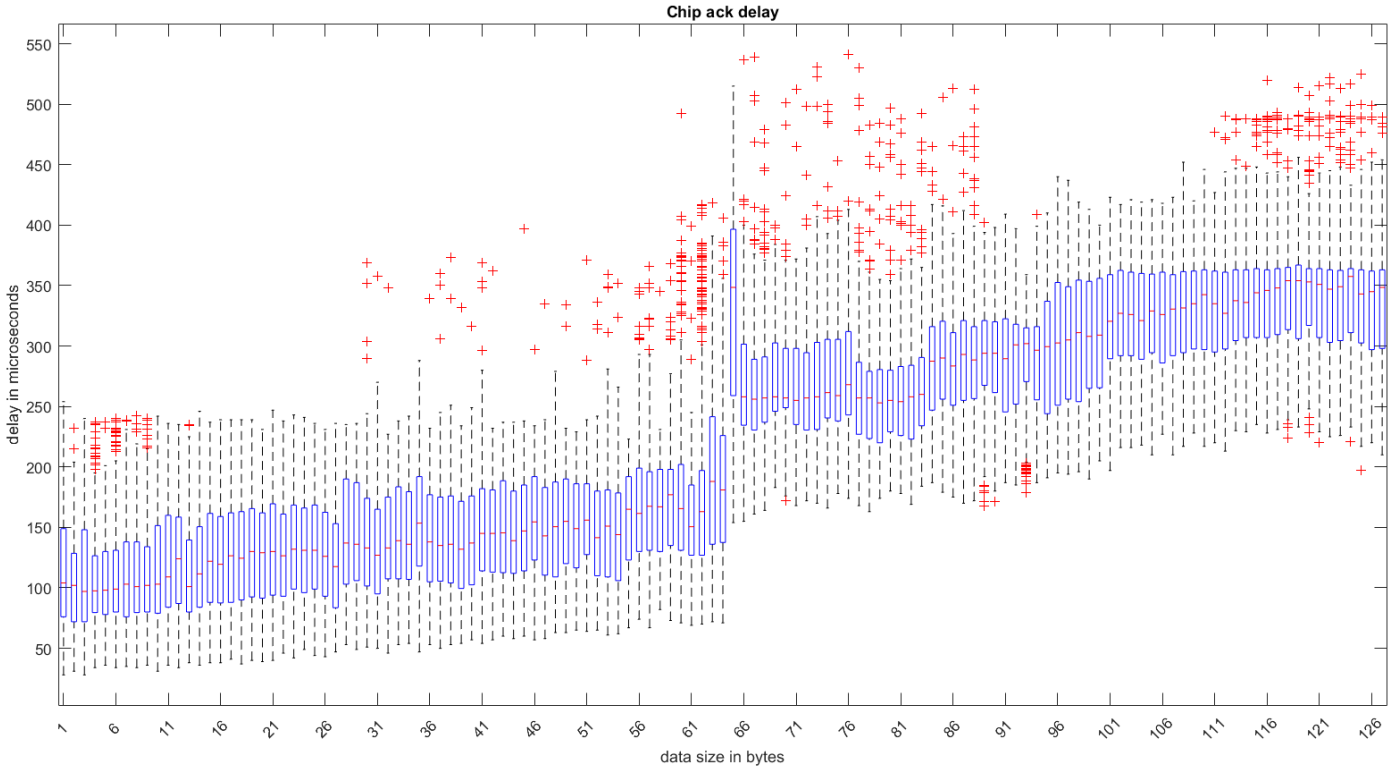


Figure 3.6: Chip ack delay

#### 3.4.2.2 USB transmission and protocol overhead delay ($T_{usb}$)

The next delay component is the USB communication delay. This delay is calculated by taking into account the speed of USB port and the protocol overheads involved. In the present experimental setup the CP2102 chip supports USB full speed, hence the throughput of USB is 12 Mbps. The CP2102 chip has bulk in and out end points with maximum packet size of 64 bytes, that means that data is transported in 64 byte sized USB packets. We have used python pyserial module to read the received data. USB protocol is a host initiated bus, Data is sent from the slave device only when the host requests data. In our experimental setup the host sends IN token(size 5 bytes) when serial.read() is called. The host application is continuously sending the serial read requests in a while loop. When the device has data, it sends data to the host. After receiving request frame host application sends USB data packets to the device in 64 byte packets. The USB transactions example is as explained in [Cra]. One USB read transaction is represented in the following figure.

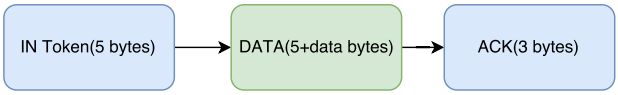


Figure 3.7: read transaction from USB slave device.

Three USB packets are exchanged for receiving request frame. IN token, DATA, ACK packet with size 3, 5+data bytes length and 3 bytes respectively with two inter packet delay of 3 bytes wide (2 $\mu$s). Total time is 12 $\mu$s, that can be attributed to USB protocol overhead. As we will see in the next section, this delay is insignificant compared to serial interface delay. In this delay, USB stack and USB controller delays have huge components.

User application running in host injects data to mote. One USB write transaction is represented in the following figure.

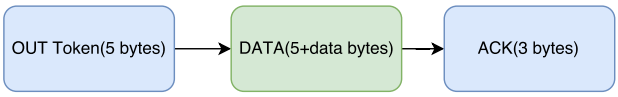


Figure 3.8: write transaction to USB slave device.

In the plot, it is observable that there is a constant delay, above which delay increases weakly until 64 bytes, this is reasonable as stack needs to copy data, calculate checksum with data.

From the plot it is evident that $T_{usb}$ reduces drastically when packet size is more than 64 bytes. This reduction is because, for bulk devices USB controllers try to achieve high throughput by forming packets of maximum size if possible, to achieve this USB controller waits for fixed time. We didn't find any value in the data sheet however from measurements(Combined delay of stack and timeout value) it approximately 180 microseconds.

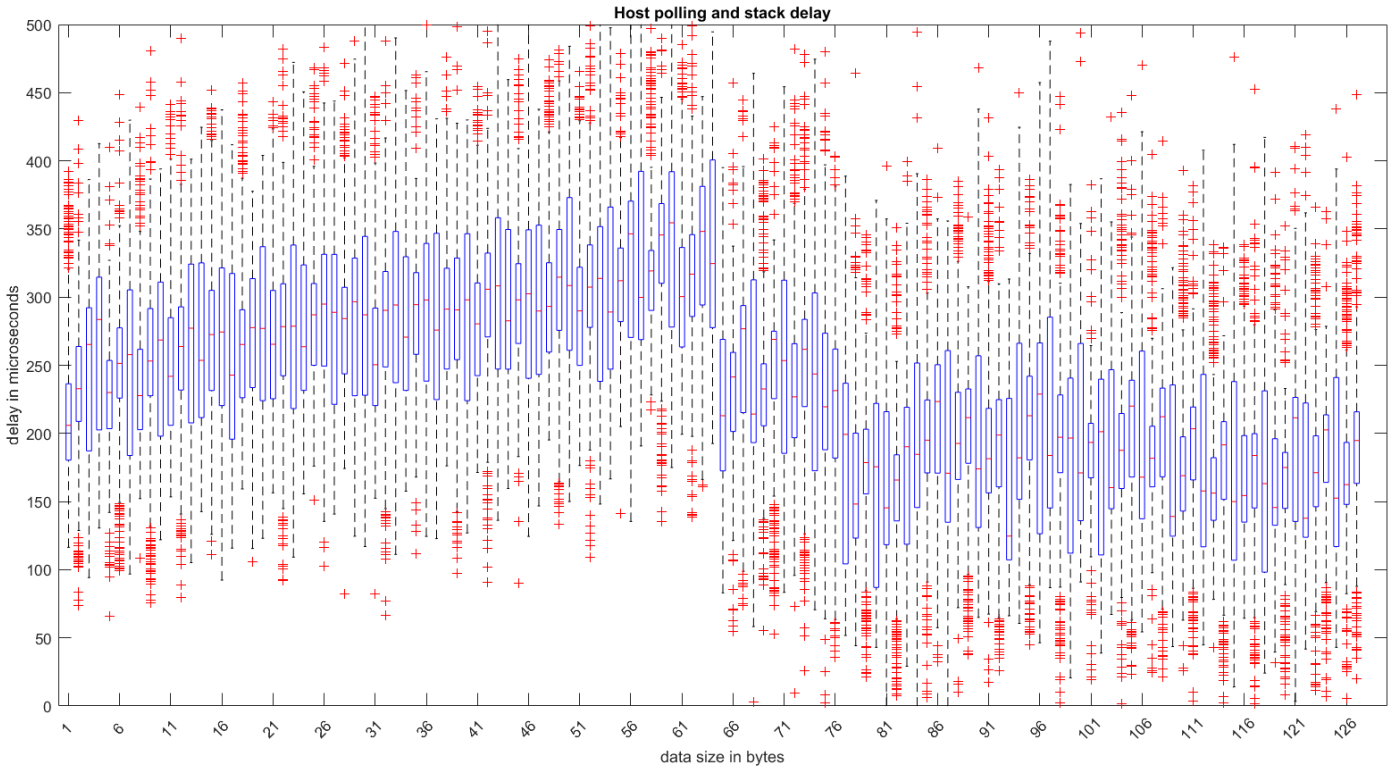


Figure 3.9: Host polling and stack delay

#### 3.4.2.3 UART transmission delay($T_{serial}$)

Lets look at the delay due to uart transmission time and receiving time. This delay is the major contributor to the latency of system, since this value is highest among all the delays. It varies linearly with the packet size. One byte transmission at 115200 baudrate requires 86.6 $\mu$s.

The following plot shows variation of uart delay versus number of uart bytes.

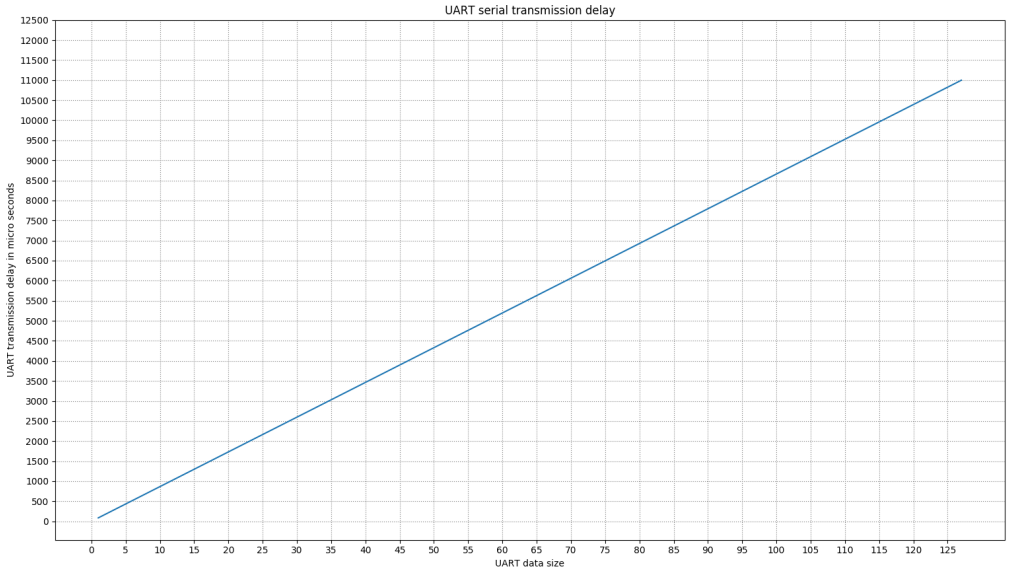


Figure 3.10: UART TX/RX delay

If we compare above plot with fig 3.5 UART tx delay's linear dependency can be clearly seen. It can be concluded that with bounded offsets of $T_{host}$ and $T_{chip}$ serial latency is a linear function of the packet size.

The total delay from east bound interface for data packet is the sum of USB stack and controller specific delay($T_{host}$), USB transmission and protocol overhead($T_{usb}$),The chip processing delay($T_{chip}$) and UART transmission delay($T_{serial}$).

## 3.5 Communication stack processing delay

Communication stack processing delay is due to the processing of data packet in the OpenWSN stack before it is stored in a queue. This processing involves adding UDP headers, compressing UDP headers, adding checksum, IPv6 headers for routing, converting packet to 6LoWPAN from IPv6, including MAC headers at the sixtop layer, then the packet is pushed in to the queue which is later read by MAC layer and transmitted.

In our experiment setup, delay is measured from the openserial driver where UDP packet is injected until it queued in the transmission buffer. This delay found to be weakly dependent on packet size.

The following figure shows the plot of stack processing delay versus UDP payload size. The Y-axis represents number of ticks taken for a particular data size (each tick corresponds

to 30 microseconds).

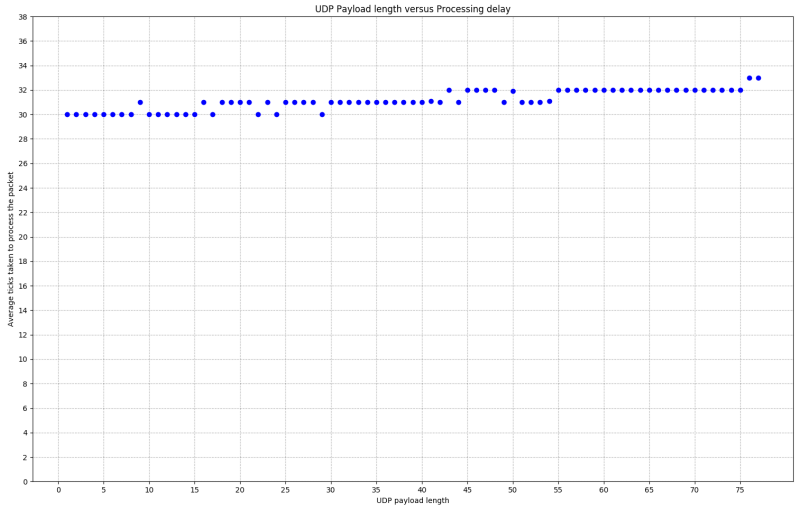


Figure 3.11: UDP packet processing delay.

## 3.6 Comparison of delay contributions

The following plot shows the different delay contributors and their contribution in the overall system latency. From the plot it can be seen that eastbound interface contributes highest among all the delay contributors.

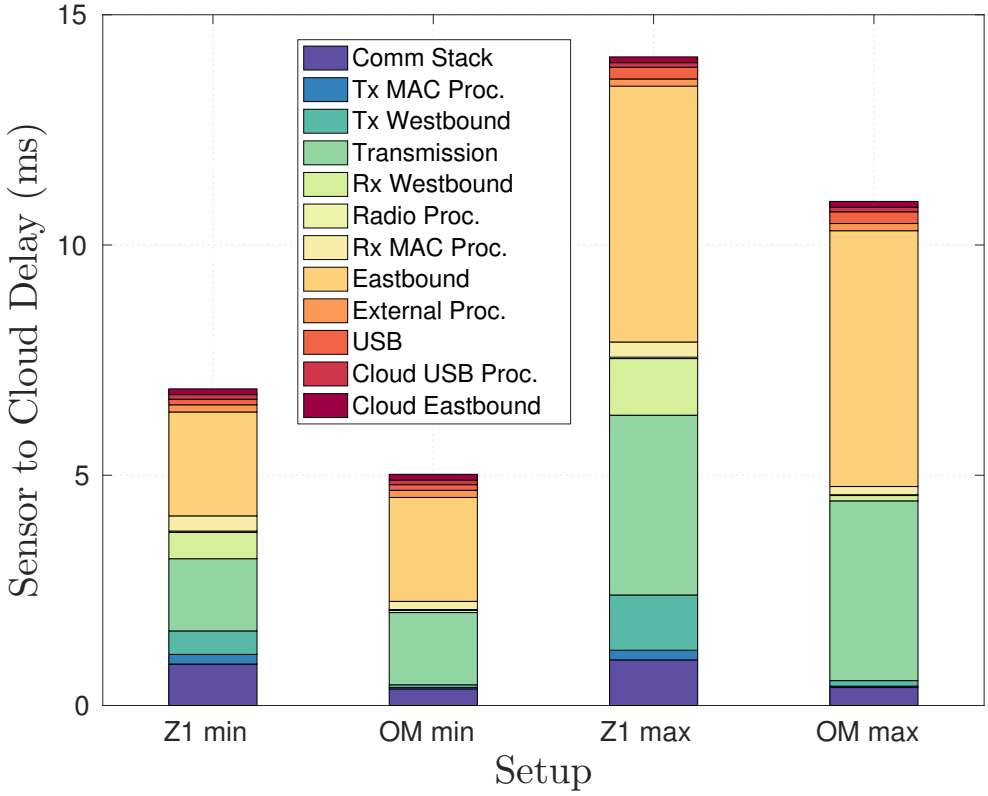


Figure 3.12: Cloud to sensor delay [H. ed]

# Chapter 4

# Conclusions and Outlook

## 4.1 Conclusion

Our goal was to evaluate latency components in wireless sensor networks. OpenVisualizer was making this very hard, since it was sending a lot of unwanted data to the host. In addition to making measurements hard it was increasing latency of the system. To make characterization easy, new eastbound interface components are developed(Openserial driver and Minimal network management python interface). Extensive latency analysis of eastbound interface is carried out. The eastbound interface was the biggest contributor among the all the delays. This was found to be the bottleneck in the overall system latency. The slot width design based on eastbound data size is presented. With the developed Openserial driver and network management module Communication stack processing delay characterization is carried out. In the last section comparison of the delay contributors are presented.

## 4.2 Future work

In the future work we would like to extend network management module to a full fledged wireless test which eases the analysis and characterization of wireless sensor networks. Until now characterization of east bound interface is carried out only software sniffing of packets. It doesn't throw light queuing delay at USB-UART chip. We would like to measure these values with hardware sniffers.

With eastbound interface characterized, with latency schedule we would like to integrate our system to simulated control system to study latency effects on stability and to explore possibility of further reduction of latency.

# Chapter 5

# Notation and Abbreviations

This chapter contains tables where all abbreviations and other notations like mathematical placeholders used in the thesis are listed.

| | |
|---|---|
| TSCH | Time slotted channel hopping |
| IPv6 | Internet protocol version 6 |
| 6LoWPAN | IPv6 over low power personal area networks |
| RPL | Routing Protocol for Low-Power and Lossy Networks |
| WSN | Wireless Sensor Network |
| CoAP | Constrained Application Protocol |
| DAG | Directed Acyclic Graph |
| DODAG | Destination Oriented DAG |
| IoT | Internet of Things |
| UART | Universal asynchronous receiver-transmitter |
| USB | Universal serial bus |
| TX | Transmission |
| RX | Reception |
| LLDN | Low latency deterministic network |

# Bibliography


[Cra] Usb in nutshell. `http://www.beyondlogic.org/usbnutshell/usb4.shtml#Bulk`. Accessed: 2017-10-12.

[GK12] Olfa Gaddour and Anis Koubâa. Rpl in a nutshell: A survey. *Computer Networks*, 56(14):3163 – 3178, 2012.

[H. ed] H. Murat Gürsu, Samuele Zoppi, Hasan Yağız Özkan, Yadhunandana R. K., Wolfgang Kellerer. Tactile sensor to cloud delay: A hardware and processing perspective. unpublished.

[Oez16] Hasan Yagiz Oezkan. Minimalistic frame structure building and bottleneck analysis for lldn with openwsn, 2016. Bachelor thesis.

[WVK+12] Thomas Watteyne, Xavier Vilajosana, Branko Kerkez, Fabien Chraim, Kevin Weekly, Qin Wang, Steven Glaser, and Kris Pister. Openwsn: a standards-based low-power wireless development environment. *Transactions on Emerging Telecommunications Technologies*, 23(5):480–493, 2012.