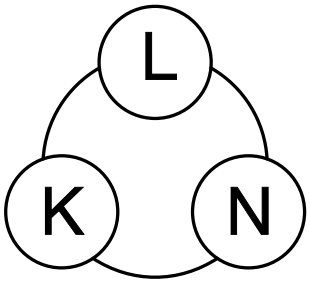

Technische Universität München
Lehrstuhl für Kommunikationsnetze
Prof. Dr.-Ing. Wolfgang Kellerer

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

This work is licensed under the Creative Commons Attribution 3.0 Germany License. To view a copy of the license, visit http://creativecommons.org/licenses/by/3.0/de

Or

Send a letter to Creative Commons, 171 Second Street, Suite 300, San Francisco, California 94105, USA.

München, 11.06.2014	
Place, Date	Signature

Abstract

This document describes the process of developing eastbound interface and latency characterization for evaluating contributors to latency in capillary networks. Although wireless communication has become integral part of our lives, still it has not succeeded in finding widespread use in industrial or automation field. It is mainly because of stringent reliability and latency requirements. However recent technological developments in wireless communication field can achieve those stringent requirements.

In this work a eastbound interface is developed to evaluate the components which cause the latency both from hardware as well software perspective. Presently many wireless sensor platforms and operating systems are available, However to the best of our knowledge we did not find any platform best suited for evaluating stringent delay and reliability of industrial communication. In this work a existing hardware platform z1 mote from zolertia and open source software OpenWSN is considered as a starting point for the development of testbed. An extendable eastbound interface is developed for controlling/injecting data to mote for communicating 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 short slot widths(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 host computer which 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 application running in the host via IPv6 sockets reads and injects data to mote for communicating to the other end of application.

Finally extensive analysis of east bound(serial) interface of the mote 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 recently industrial communication was mainly through wired networks such as Ethernet,Modbus etc. The reason being strict reliability, latency and robustness requirements in industrial environments.

Recent innovations in wireless technologies and hardware has made wireless chips affordable,cheap simultaneously 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 monitored remotely without tedious and costly wired networks. Wireless nodes being easily installable,replaceable become an attractive option. With all these advantages, still there are problems which needs to solved. For example to satisfy stringent latency and reliability requirements, new type of medium access methodologies have to be developed, implementations need to optimized for achieving low latency and to maintain network reliability new routing algorithms have to developed by keeping in mind these stringent requirements while defining the objective function. The stake in manufacturing environments is very high. a network disconnection might result in a loss of billions.

In this internship, work is carried out to develop eastbound interface and its characterization for Industrial wireless testbed to evaluate wireless sensor networks for latency and reliability. To develop such a testbed OpenWSN software stack is considered. It provides a open source implementation of new IEEE802.15.4e 'Time synchronized Channel hopping' medium access standard which achieves high reliability through frequency agility(Channel hopping) to provide reliablity in irrespective of channel congestion and low power through time synchronization with other nodes in the network. The development of platform is done on the commercially available of the shelf hardware platform Zolertia Z1(Framework is not tied to any hardware as long as the platform supports computing and radio re-

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

quirements). In research internship main focus was developing testbed for analyzing delay contributors, Low latency network setup and characterization of the delay contributors in the east bound interface(host to mote serial communication) of the network.

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

In chapter 3 we details of our implementation are presented. First part explains the design of openserial driver module and details of Minimal network management python module. Second part describes the LLDN schedule construction. Third part briefly describes modification of external MAC. In the fourth part presents extensive characterization of east bound interface of is presented. At last in fifth part communication stack processing delay for OpenWSN stack is presented.

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

Chapter 2

Background

The chapter discusses the basic terminologies and approach used in this internship, followed by the research question of this internship, the related work and various existing tools were found to be suitable or unsuitable to achieve the goals.

Industrial domain has opted wireless sensor networks over traditional wired communication due to several attractive features of IWSN's such as self-organization, rapid deployment, flexibility. the explosive growth that has occurred in the use of wireless sensor networks in a variety of applications. As wireless technology can reduce costs, increase productivity, and ease maintenance. However the advantages come with some challenges which needs to 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 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 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 perceived 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 vary 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 the 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 and, hence, need to be integrated with the Internet Protocol (IP) architecture.

2.1 OpenWSN Stack

OpenWSN is open source internet of things stack based on highly reliable time slotted channel hopping based 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.

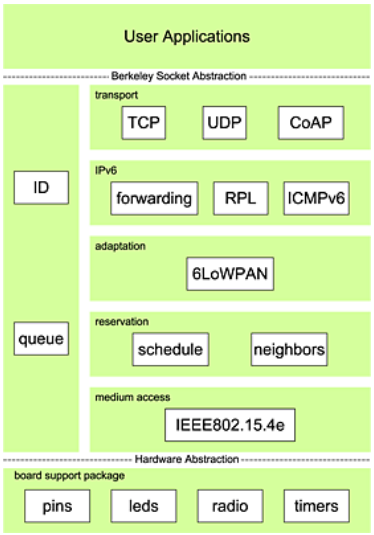

Figure 2.1: OpenWSN stack[WVK+12]

2.2 TSCH

Reliability and deterministic behavior is required in industrial communication with low power consumption, since nodes are running with scavenged or battery power. TSCH

is medium access mechanism inspired from WirelessHART and ISA100.11a. TSCH(Time slotted channel hopping)provides both low power and high reliability. In TSCH mechanism nodes in the network are time synchronized with the super frame and cells 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 the 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 mostly require high determinism,reliability and low latency. LLDN(Low Latency Deterministic Network) standard of IEEE considers latency as main QoS element.

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

2.4 Problem description

OpenWSN is an opensource implementation of a 6tisch standard coupled with internet of things standards such as 6LoWPAN,RPL and CoAP allows low power ultra reliable communication, However design is not optimal for low latency communication and also 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,API's 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 so in our implementation we are concentrating more on these parameters. With wireless testbed it is easy conduct experiments and evaluate different algorithms.

In the process of developing low latency wireless testbed first step is naturally setting up of the low latency network in which latencies can be measured or characterized accurately, The road block for this was openvisualizer. which is necessary for setting of WSN. however OpenWSN was making latency measurement/debugging difficult. So step explored is to figure out a way to set up WSN network without OpenVisualizer, Which involved understanding the type functionalities OpenVisualizer is doing and which functionalities are absolutely necessary.

Through thorough investigation of OpenVisualizer code following functionality found to be implemented.

1. Maintaining routing table for nodes in the network for downward traffic using source routing headers.
2. Communicating motes at the exact super serial RX and TX frame slots.
3. Controlling/configuring the parameters of sensor network.
4. Packets read from TUN/TAP interface and then injected to mote via serial after converting to 6LoWPAN format with source routing header.
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), hence making super

frame long and resulting in more latency.

6. Modules for redirecting packets received from sensor network to wireshark for debugging.
7. HDLC protocol implementation.

All this data is not sent to host without host asking for it. This is lot of traffic which causes significant latency in overall system.

In the above findings, functionalities which contributed most to latency and made latency characterization is the 5^{th}, 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 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 PC side tool replacing OpenVisualizer which has minimal functionality necessary for maintaining the network. This PC side tool is developed in Python. In addition to maintaining network, it should ease measurement of latency in different components of system.

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

3.1.1 Open serial driver design

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

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

The scheduling of the serial driver is carried out according to super frame timing. openserial driver api's `openserial_startOutput`,`openserial_startInput`,`openserial_stop` are the function which do all the work.

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

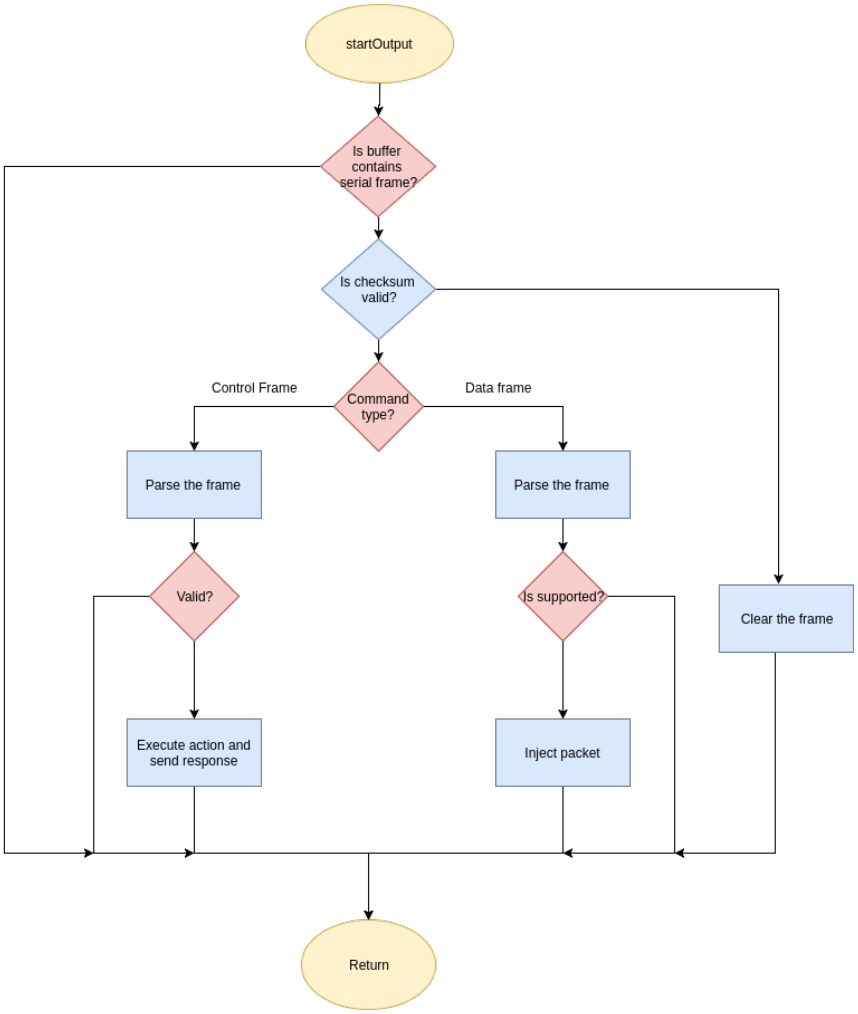

Figure 3.1: Serial frame processing

`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 API's were implemented in OpenWSN originally to schedule the openserial component from MAC layer. In the present design too scheduling is kept same as the original one. Except modification of MAC layer related to serial communication as explained in the last subsection.

Figure

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

Buffer management:
Open serial driver manages the data in two circular buffer implementations TX_BUFFER and RX_BUFFER of size 256 bytes. When SERIAL_RX slots are scheduled, hosts send data via uart. Once byte received in the interrupt handlers data is pushed in to SERIAL_RX buffer. This buffered data is processed in 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 buffering data which needs to sent via serial to host computer. Whenever mote wants send data(Eg:Sending packet received or debug information) It call `openserial_printf` This function is part of open serial driver which buffers this data to TX_BUFFER. When SERIAL_TX slot is scheduled this data is popped from TX_BUFFER and transmitted via UART. As will be explained later in this chapter, With aim reducing the latency, slot widths in TSCH schedule is reduced as minimal as possible. During this research internship slot width 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 sent via particular baudrate with following simple equation.

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

For example at baudrate of 115200 and 6ms slot width, at max 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 taken into account.

Sample serial frames(Data frames, command frames) In the openserial driver to make implementation neat two types of serial frames are implemented, namely data serial frames and command serial frames which are used for network management purpose. For example serial frames which are used for injecting UDP packet data belong to serial data frames category, Serila frames which are used for getting TSCH schedule belong to commnd serial frame category. This type modularization helps to modularize the serial frame processing logic in openserial driver.

To make serial frames packet 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 makes configures the mote as DAGroot. The first byte indicates beginning of the frame, second byte length of the packet excluding `0x7e` , 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 too follows the same protocol explained above. 3rd byte represents command category as Data related fourth byte represent the specific command types 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 that is third byte in very useful for modularizing the processing logic in open serial driver. When processing the packet received, if the open serial driver checks this type, delegates the processing of this packet to respective module(Command processing category or data processing category).

3.1.2 Network management module

After open serial 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 this 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.

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 this transmits the advertisement packets about presence of the network other nodes join this network. therefore one node has to be dagroot the functionality is achieved by sending a serial command frame.

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 only, When a host wants to know stack information corresponding command is sent, network statistics is returned in the response frame.

Latency plays a very significant role in industrial communication, Measuring latency is required, it may be latency due to communication stack,MAC schedule or serial etc. To make these measurements easy, Few options are provided in the framework where stack injects the data, latency info is taken from stack sent through response frame this information is saved in json format for further processing.

In OpenWSN stack upward routing is implemented RPL[GK12] which based on DOADG's. RPL works for upward routing and it is specifically designed for data collection networks. OpenWSN uses achieves downward routing using source routing mechanism, the which needs to routed contains the route, Each node in the path looks at the packet forwards according to route specified. DAG root runs only the MAC layer in OpenWSN stack. When packet needs to sent downward from DAG root, that packet should contain routing information for this DAGroot must be aware of the complete network, This is achieved in OpenWSN with DAO messages. All the nodes of network send information about their parents to DAGroot, this information is relayed to host application where routing table is built before injecting packet to DAGroot host side application creates source routing header. This was in OpenVisualizer too, Same implementation is adapted to Network management module.

Handling schedule is one more feature required for adjusting the schedule according to traffic/latency requirements. Because of this reason three specific commands are implemented view schedule,add/remove RX or TX slots.

In host computer industrial control system(Either sensor/controller) is running it sends calculated/sensed data via sockets this data needs to be received,converted to 6LoWPAN packet. This functionality too is implemented as as part of this python module.

Finally 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 the two.

Figure 3.2: Experimental setup

3.2 LLDN schedule construction

To achieve low latency as MAC layer is modified, 802.15.4e MAC mechanism allows synchronization with ack's too. However this synchronization based on ack's make slot width long. In turn increases latency. To avoid this ack's are removed from the schedule and synchronization for of network nodes is done only through beacon frame's with their frequency increased. Sample schedule is shown below. [Oez16]

In the schedule TXRX frame 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.

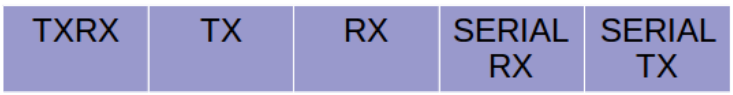

Figure 3.3: schedule

3.3 Modified OpenWSN External MAC

External MAC is a mechanism to choose when data has to be sent from user application running in host computer to mote and vice versa. Since the mote has single micro controller this communication time also should be part of super frame schedule. In order to do that serial communication timing is accommodated in schedule through serial RX and TX slots, These slots are used for communicating with host computer running user application. The mechanism is as explained below. In the schedule as soon serial RX 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 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 request frame is sent to host. In this design modification data can be sent from host computer in every SERIALRX slot.

3.4 Eastbound delay characterization

This document describes the characterization of delays involved in the eastbound interface of wireless sensor network setup. For the characterization of delays open source IoT stack OpenWSN is considered.

3.4.1 Setup

In the experimental setup Zolertia-Z1 mote running OpenWSN is considered. Mote is connected to 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 a ideal communication protocol for achieving real time communication especially if the slave device is bulk type device. USB controller schedules the bus access to the bulk devices based bus availability and latency is not guaranteed. This latency varies depending on how many devices connected to the USB controller and also on the type of end points they have. For example: If more devices with interrupt and isochronous end points are connected then bulk devices will be deprived from the bus access, since those(interrupt and isochronous) end points have higher priority than bulk end points. When the mote sends data to host there is one more additional delay involved called USBtimeout value of chip. This delay caused because 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 USBtimeout duration request frame is delayed by USBtimeout duration. This delay affects the overall latency of the system.

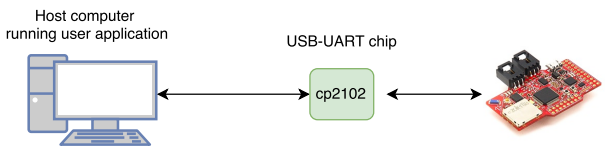

Figure 3.4: experimental setup.

3.4.2 Evaluation

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

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

The T_{host} specific delay is due to USB driver and USB controller. In the experimental set up this value is found to vary as plotted in 3.6 on page 19.

The protocol overhead delay is due to inherent nature of USB protocol. This delay comprises of the time involved in sending IN,OUT,ACK packets plus data transmission time. Once the data from host is received at the USB to serial converter chip. This data is transmitted to mote through serial communication. This delay is highest among all 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 the plot total eastbound delay versus data size, The delay is almost linear function of data size. This dependency is driven mainly by T_{serial}

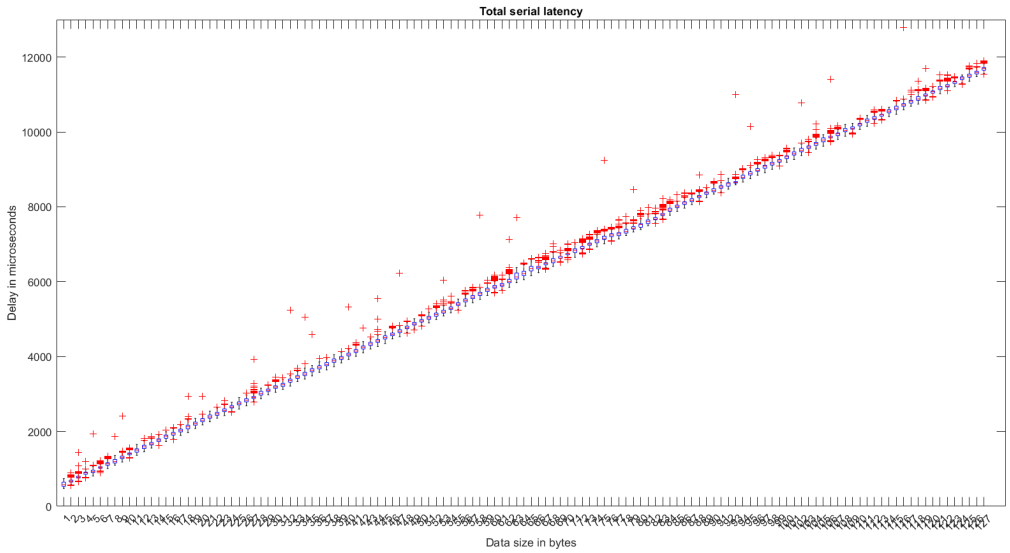

Figure 3.5: Total serial latency

At last, chip delay. this delay is measured by taking USBmon logs to identify how much delay chip is taking for sending ack's after receiving the USB packet. This delay also found to be significant. Please see the plot 3.9 on page 21.

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, it is obtained from measurements. It can varies from based on the packet

3.4.2.1 Chip ack processing delay

This delay is the time taken from CP2102 USB-UART chip to send the USB ack after USB packet is sent, With the measurements from USBmon it is found to very significant. This values found to be not constant instead stochastic except USB timeout value.

As discussed earlier cp2102 chip has a parameter called USBtimeout. This is the time after which cp2102 chip starts sending received uart bytes as usb packets, This value is dependent on the baud rate. At 115200 baud rate USBtimeout value is 156 μs. This delay is required when data is sent from mote to host, Since waits for 156 μs before sending data to host. This value is constant hence not included in chip ack delay plot.

Chip ack processing delay is data size dependent with data size this value increase with significant increase when packet bigger than 64 bytes. This increase is because chip has to wait for another USB data packet before sending the ack to host. Chip ack delay values are plotted in figure 3.6 on page 19.

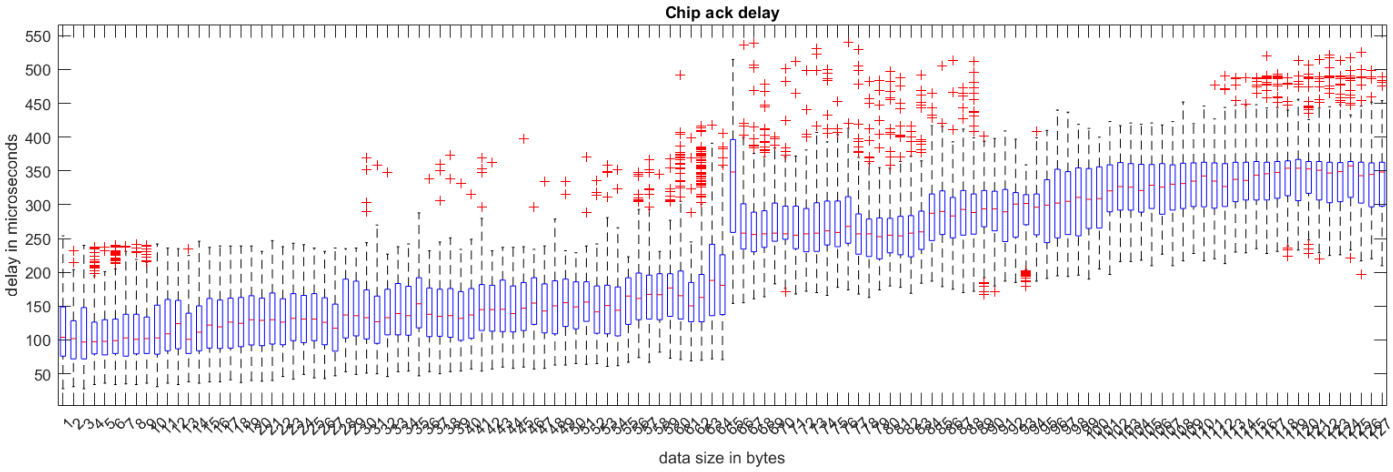

Figure 3.6: Chip ack delay

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

The next delay component is USB communication delay. This delay is calculated by taking into account speed of USB port and protocol overheads involved. In the present experimental setup 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 data is transported in 64 byte USB packets. We have used python pyserial module to read received data. USB protocol is host initiated bus, Data is sent from the slave device only when host requests data. In our experimental setup host sends IN token(size 5 bytes) when serial.read() is called. Host application is continuously sending the serial read in infinite while loop. When device has data, it sends data to host. After receiving request frame host application sends USB data packets to the device in 64 byte packets. One USB read transaction is represented in the following figure.[Cra]

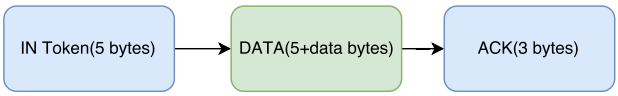

Figure 3.7: read transaction from USB slave device.

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 μs). Total time is 12 μs can be attributed to USB protocol overhead. As we will in the next section this part has very compared serial interface. In this delay USB stack and USB controller delay have huge components.

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

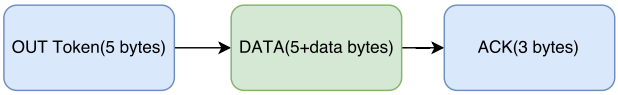

Figure 3.8: write transaction to USB slave device.

In the experimental setup packet with 12 bytes of user data is considered, including the lowpan headers,source routing header and control bytes of openserial module total packet size becomes 54 bytes. Three packets are exchanged in write transaction with packet size 5,(5+54) and 3 bytes respectively with two inter packet delay of 3 bytes results in 49 μs.

In the above plot, It is observable that there is constant delay, Above which delay increases weakly until 64 bytes this is reasonable as stack needs make operations 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 packet if possible, To do 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.

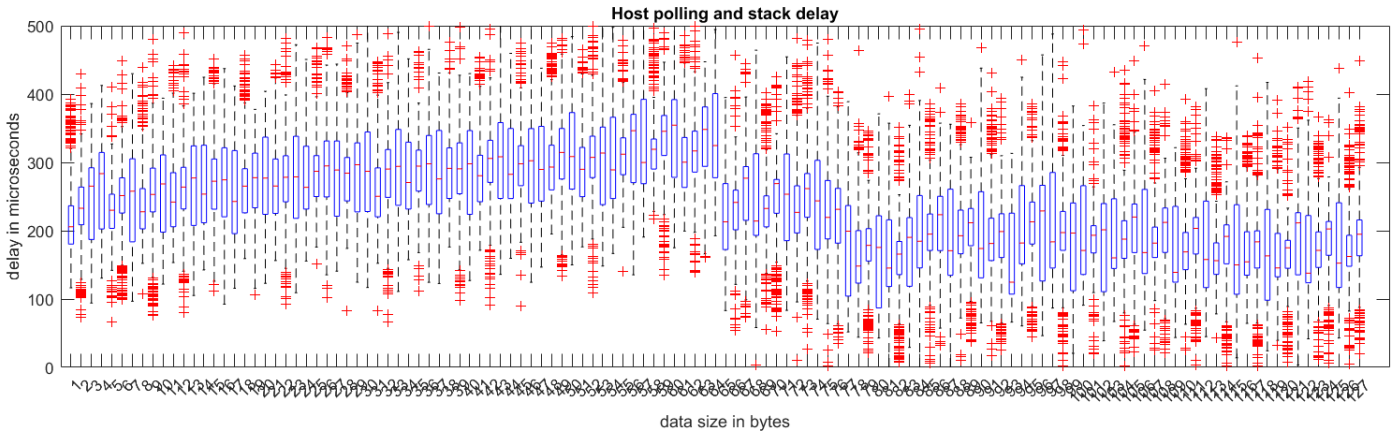

Figure 3.9: Host polling and stack delay

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. varies linearly with the packet size. For transmitting one byte at 115200 baud rate 86.6 μs is needed.

Following plot shows variation of uart delay versus number of uart bytes.

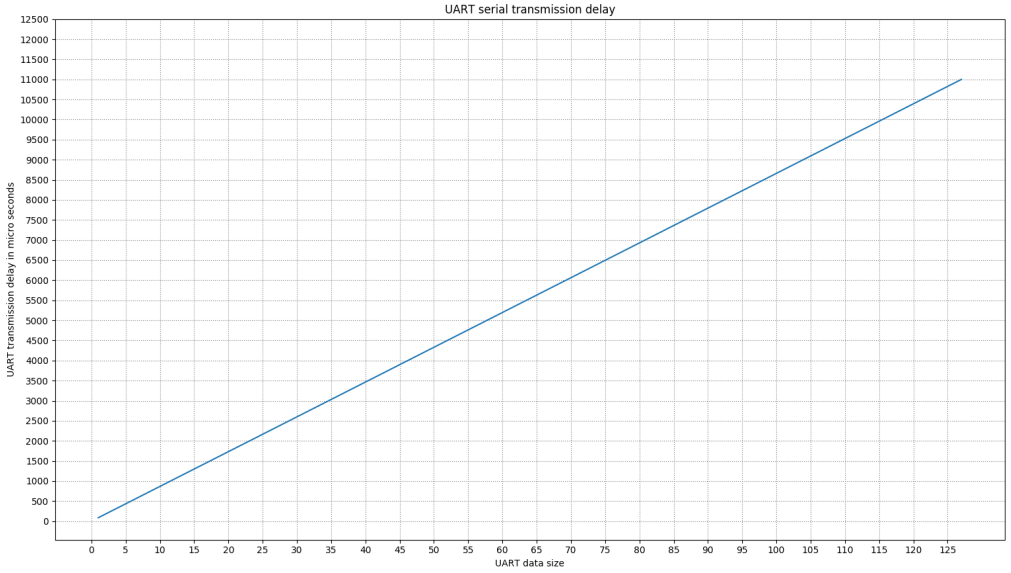

Figure 3.10: UART TX/RX delay

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

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

Similar estimation and measurement is carried out for a serial packet of length 90 bytes. Only difference in estimation from previous calculation is that data needs to be sent in two USB packets in chunks of 5(header) + 59(data) and 5(header)+31(data) bytes so extra 5 byte header of second packet and 3 byte interpacket delay needs to taken into account. Results are shown in the following table.

3.5 Communication stack processing delay

Communication stack processing delay is due the processing of data packet in 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 packet is pushed in to queue which is later read by MAC layer and transmitted.

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

Following figure shows plot of stack processing delay versus UDP payload size. Y-axis represents number of ticks taken for a particular data size(each tick corresponds to 30 microseconds).

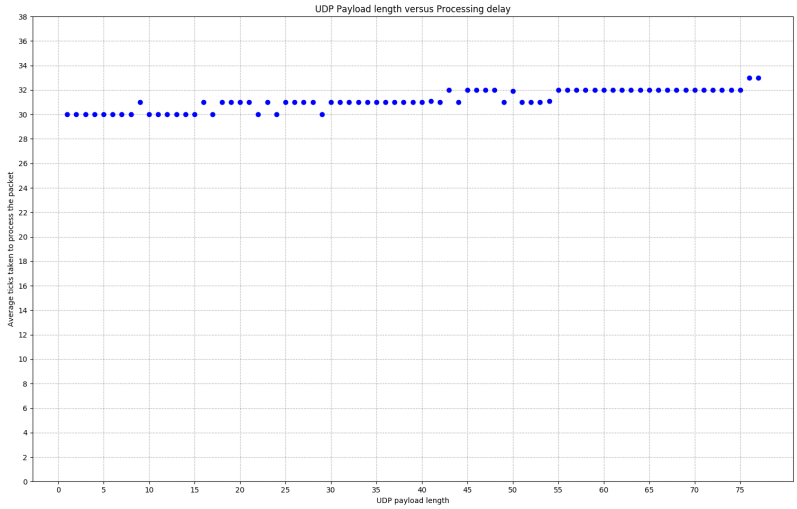

Figure 3.11: UDP packet processing delay.

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. To make characterization easy new eastbound interface components are developed(Open serial driver and Minimal network management python interface). Extensive latency analysis of eastbound interface is carried out. This found to be bottleneck in overall system latency. Slot width design based on eastbound data size is presented. At last Communication stack processing delay characterization is carried out.

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.

Appendix A

The appendix may contain some listings of source code that has been used for simulations, extensive proofs or any other things that are strongly related to the thesis but not of immediate interest to the reader.

Appendix B

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.

[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.