O. YILMAZ

ETHERNET BASED REAL TIME COMMUNICATIONS FOR EMBEDDED SYSTEMS

OZAN YILMAZ

METU 2010

MAY 2010

ETHERNET BASED REAL TIME COMMUNICATIONS FOR EMBEDDED SYSTEMS

A THESIS SUBMITTED TO
THE GRADUATE SCHOOL OF NATURAL AND APPLIED SCIENCES
OF
MIDDLE EAST TECHNICAL UNIVERSITY

BY

OZAN YILMAZ

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR
THE DEGREE OF MASTER OF SCIENCE IN
ELECTRICAL AND ELECTRONICS ENGINEERING

MAY 2010

Approval of the thesis:

ETHERNET BASED REAL TIME COMMUNICATIONS FOR EMBEDDED SYSTEMS

Submitted by **OZAN YILMAZ** in partial fulfillment of the requirements for the degree of **Master of Science in Electrical and Electronics Engineering Department, Middle East Technical University** by**,**

Prof. Dr. Canan Özgen
Dean, Graduate School of **Natural and Applied Sciences** _____

Prof. Dr. İsmet Erkmen
Head of Department**, Electrical and Electronics Engineering** _____

Asst. Prof. Dr. Şenan Ece Schmidt
Supervisor, **Electrical and Electronics Engineering Dept**., **METU** _____

Examining Committee Members:

Prof. Dr. Semih Bilgen
Electrical and Electronics Engineering Dept., METU _____

Asst. Prof. Dr. Şenan Ece Schmidt
Electrical and Electronics Engineering Dept., METU _____

Prof. Dr. Hasan Güran
Electrical and Electronics Engineering Dept., METU _____

Assoc. Prof. Dr. Cüneyt Bazlamaçcı
Electrical and Electronics Engineering Dept., METU _____

Yusuf Bora Kartal, M.Sc. ASELSAN. A Ş _____

Date: 10.05.2010

I hereby declare that all information in this document has been obtained and presented in accordance with academic rules and ethical conduct. I also declare that, as required by these rules and conduct, I have fully cited and referenced all material and results that are not original to this work.

Name, Last name : Ozan YILMAZ

Signature :

ABSTRACT

ETHERNET BASED REAL TIME COMMUNICATIONS FOR EMBEDDED SYSTEMS

Yılmaz, Ozan
M.Sc., Department of Electrical and Electronics Engineering
Supervisor:Asst. Prof. Dr. Şenan Ece Schmidt

May 2010, 96 Pages

Fast paced improvement of Ethernet technology has also received attention in the industry field like it did in other fields and ways of usage have started to be studied. As it is understood that the standard Ethernet protocols cannot be used due to the unsatisfied real time requirements, industrial and academic researchers have started to develop solutions to overcome this deficiency. In this thesis, the real hardware adaptations of Real Time Ethernet and RTXX protocol algorithms are implemented and their behaviors on the hardware are observed. Each parameter that affects the system's real time behavior is individually examined and the solution proposals are discussed.

Keywords: Industrial communication, real-time, Ethernet

ÖZ

GÖMÜLÜ SİSTEMLER İÇİN ETHERNET TABANLI GERÇEK ZAMANLI HABERLEŞME

Yılmaz, Ozan
Yüksek Lisans, Elektrik ve Elektronik Muhendisliği Bölümü
Tez Yöneticisi: Y. Doç. Dr. Şenan Ece Schmidt

Mayıs 2010, 96 Sayfa

Ethernet teknolojisinin hızla gelişmesi diğer alanlarda olduğu gibi endüstri alanında dikkatleri çekmiş ve kullanım yolları aranmaya başlanmıştır. Gerçek zamanlılık ihtiyaçları nedeniyle standard Ethernet protokollerinin kullanılamayacak oluşunun anlaşılmasıyla sanayii ve akademik kaynaklar bu eksikliği kapatmaya yönelik çözümler üretmeye başlamışlardır. Bu tez çalışmasında Gerçek Zamanlı Ethernet ve RTXX Protokol algoritmalarının gerçek donanım uyarlamaları gerçekleştirilmiş ve donanım üzerinde davranışları izlenmiştir. Sistemin gerçek zamanlılığını etkileyen her bir parametre ayrı ayrı incelenmiş ve çözüm önerileri tartışılmıştır.

Anahtar Kelimeler: Endüstriyel haberleşme, Gerçek Zamanlılık, Ethernet

To My Parents

ACKNOWLEDGEMENTS

I am in great debt to Asst. Prof. Dr. Şenan Ece Schmidt, my research advisor; she labored hard for two years to ensure I would accomplish my research goals.
I wish to thank TEKTRONIK A.S for giving me the opportunity of continuing my education
I would like to thank my parents, my sister and my fiancee for their patience and trust throughout my thesis.
I wish to thank to my friend Koray OKŞAR and my colleagues for their valuable support.

TABLE OF CONTENTS

LIST OF TABLES

TABLE OF FIGURES

TABLE OF CODES

LIST OF ABBREVIATIONS AND ACRONYMS

ACPI	Advanced Configuration and Power Interface
API	Application Programming Interface
APIC	Advanced Programmable Interrupt Controller
CAN	Controller Area Network
CM	Communication Model
CMOS	Complementary Metal Oxide Semiconductor
CSMA/CD	Carrier Sense Multiple Access with Collision Detection
DES	Discrete Event System
FPGA	Field-Programmable Gate Array
FTP	File Transfer Protocol
GPS	Global Positioning System
HPET	High Precision Event Timer
IEEE	The Institute of Electrical and Electronics Engineers
IP	Internet Protocol
IPC	Inter Process Communication
MAC	Medium Access Controller
MTSD	Master to Slave Delay
NRT	Non Real Time
PHY	Physical
PIT	Programmable Interval Timer
PLC	Programmable Logic Controller
POSIX	Portable Operating System Interface for Unix

PTP	Precision Time Protocol
RT	Real Time
RTC	Real Time Clock
RTE	Real Time Ethernet
STMD	Slave to Master Delay
TDMA	Time Domain Multiple Access
TSC	Time Stamp Counter

CHAPTER 1

INTRODUCTION

Nowadays, the use of communication networks is common at all levels of industrial automation systems in the process control and manufacturing industry. Amongst others, communication networks are employed to enable the acquisition and distribution of sensor and actuator data on the device or machine level, the coordination of processes among distributed controller devices on the cell and subsystem level, and the production scheduling, monitoring and management at the system and factory level.

The traffic that has to be carried by such industrial communication networks has different characteristics. While hard real-time applications such as closed-loop control demand guaranteed bounds on the delivery times up to under 1ms and soft RT applications in automation and manufacturing require delivery times in the order of 10ms, there are non-real-time (nRT) processes such as diagnostic monitoring or maintenance without stringent timing requirements.

In the past, the communication requirements were met by different network types at different levels of the automation hierarchy. On the one hand, Fieldbuses were developed for the frequent and timely communication of small data packets as required on the lower levels of the automaton hierarchy. On the other hand, Ethernet is well-suited for the less time-critical operations on the higher levels of the automation hierarchy.

In the recent years, there is a strong tendency to replace fieldbusses by Ethernet due to various reasons. Since fieldbusses cannot cope with the increasing data volumes on industrial networks, they must be upgraded in order to support higher data rates. However, considering the relatively small market for such devices, the development costs are disproportionate. In contrast, Ethernet provides high speeds at low costs due to its pervasiveness in home and office environments. The main obstacle for the direct use of Ethernet for the time-critical data transmission on the lower levels of the automation hierarchy is its lack of RT support due to the nondeterministic carrier sense multiple access with collision detection (CSMA/CD) arbitration mechanism . Hence, there is an ongoing effort to provide Ethernet-based industrial network solutions with RT support and thus converge to a single network technology on the different levels of the automation hierarchy.

Different approaches for the development of Real-time Ethernet (RTE) are pursued in both industry and academia. In order to achieve a deterministic timing behavior without collisions on the medium, such approaches propose modifications and additions to the network protocol stack of conventional shared-medium Ethernet or employ switches. Industrial protocols that belong to the first category combine the use of standard Ethernet hardware with master-slave communication and the definition of pre-specified periodic sending instants. There are also protocols which are based on full-duplex switched Ethernet with a specialized prioritization scheme, or which are designed for customized controller or switch hardware.
A common feature of these protocols is that they provide real-time support on Ethernet by a static configuration of the possible sending instants or the RT-bandwidth allocated to each networked controller device. However, it is not considered that the communication requirements of automation applications dynamically change depending on the operating condition of the application .

In this thesis the implementation and performance evaluation of RTXX, an Ethernet-based industrial communication protocol is presented. RTXX is designed to be implemented with TDMA over shared Ethernet and it exploits the determinism of the

industrial applications that communicate over the network. In this protocol, each node computes the forthcoming communication requirements of the application and informs the other nodes in the network accordingly. Consequently each node is able to compute independently the order of medium access among the nodes for a certain number of coming time slots.

The implementation of RTXX protocol presented in this thesis includes the realization of TDMA over Ethernet and the required time synchronization mechanism among the nodes. In addition the decision mechanism for the medium access and the required information exchange is incorporated. The correctness of the operation and the satisfaction of the real time requirements are verified with an experimental study.

The remainder of this thesis is constructed as follows. In Chapter 2 we review the Real Time Communication Protocols in the literature. Characteristic requirements for Industrial applications is defined and discussed during this chapter. In Chapter 3, architecture and characteristics of the RTXX Protocol is defined, a sample communication model for an industrial bottle filling machine is illustrated. In Chapter 4 Time Domain Multiple Access based Real Time Ethernet Implementation is proposed and discussed in detail. In addition, the requirements for software and hardware implementations presented in detail and solutions are discussed. Time synchronization mechanisms were also included in this chapter. In Chapter 5 RTXX Software Implementation is proposed and related software modules discussed in detail. Interface between TDMA Layer and RTXX Protocol Implementation also explained in this chapter. User application interfaces for RTXX Protocol is defined and presented in this chapter. Chapter 6 contains the experiments and discussion about measurement results. Chapter 7 concludes the thesis.

CHAPTER 2

REAL-TIME INDUSTRIAL COMMUNICATION PROTOCOLS

Industrial control systems that communicate over a network is formed by the sensors, the controllers (programmable logic controllers), industrial PCs and actuators [1] which are coordinated over a communication channel. The industrial control applications which are getting more complex and large scale and industrial control equipment which are being manufactured with computer and network support has made the industrial control systems that communicate over network an important industrial and academic research topic. For these systems different industrial communication networks are developed for the last twenty years. [1],[2],[3],[4].

In industrial communication networks messages should be transmitted which have varying purposes and properties. [2]

T1) Equipment level data transfer between sensors, controllers and actuators: Continuous or sampled data are usually sent periodically and with time constraints. Example: The data that is collected from the speed sensor in the servo driver is sent to the controller, output current value is sent from the controller to the actuator.

T2) Messages that are at the supervisory control level: Communication between system components that are at different hierarchical levels is required according to the hierarchical organization of the controllers and the controlled systems. Mostly, data which necessitates event based and deterministic reaction times is sent. Because the system's behavior changes at discrete times, the system's next condition and the messages that will be sent in this condition can be known beforehand by using the condition that the system is in and the system's dynamic model. Example: According

to the model of a controller system that controls two machines, after the work finish event of the first machine occurs it sends a message to the second machine to make it start working. This is stated in the system model in which the message will be sent together with the first machine's finish work event.

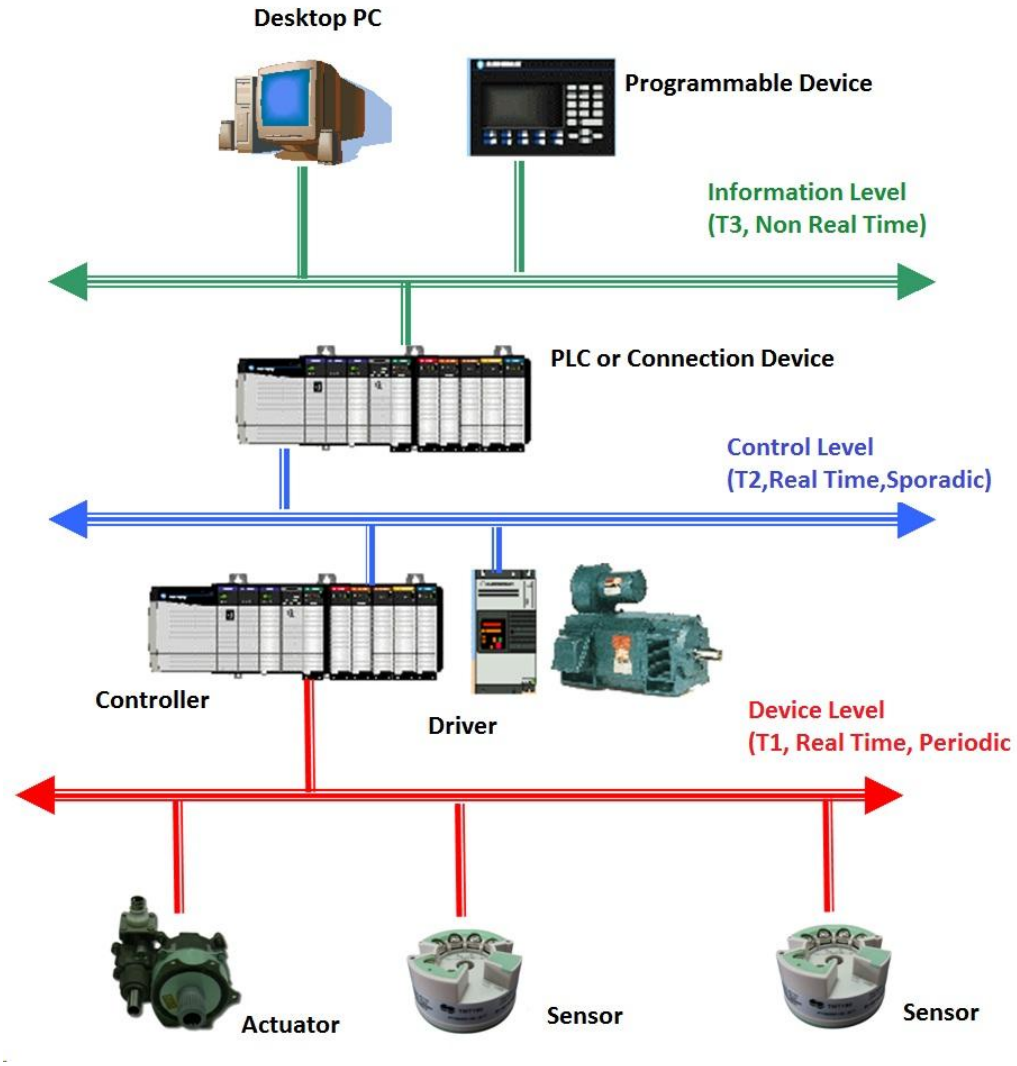

Figure 1: Different Levels of Communication over Industrial Networks

T3) Diagnostic data and remote control oriented applications: Mostly moved as non real time and event based communication.

When these traffic types are examined, four impo rtant requirements in the industrial communication systems can be identified:

1) Real time traffic transfer: The transferring of the messages (T1, T2 traffic types) before a determined deadline after their creation.

2) Concurrent communication: The requirement of the nodes in the network to have a shared time base for providing a real time traffic transfer. (T1, T2 traffic types)

3) Dependability: The dependability support for the fault and failure conditions in the industrial control applications.

4) Non real time traffic support: The transport of these kinds of messages without sacrificing the effectiveness of the real time messages. (T3 traffic type)

Providing the real time guarantees in industrial communication networks requires the delivery of the messages with no delay at least. Likewise the dependability support should be according to the worst conditions to provide the desired error possibility. For these reasons, total capacity requirement in industrial networks and the capacity allocation are calculated according to the usually unrealistic assumptions such as sending of all the messages at the same time.

The preliminary communication networks for the industrial environments like CAN, Lon Works and Profibus are started to be used nearly twenty years ago [5]. These field buses were proprietary, expensive, hard to develop, not compatible with each other and had not fastly attuned to the changing industrial applications. On the other hand, the simple, cheap and fast Ethernet which is prevalent in home and office environments is an important candidate for industrial communication. Despite these superiorities, on Ethernet, the messages that are sent at the same time are colliding and being resent at random times. Because of this reason, the standard Ethernet cannot provide deterministic network access and cannot support real time dependable

communication. Academic and industrial studies are being conducted for the advancement of real time Ethernet (RTE) since recent years.

Real time control applications have necessities like: receiving reactions at restricted times, minimum deflection from periodicity of events which need to be periodic and protection of the time sequences of the events. For these requirements to be satisfied in the distributed systems, concurrency and temporal consistency should be ensured between system components. [3],[4]. IEEE 1588 which is a new protocol is designed especially for synchronization of real time systems over small distributed networks like industrial control systems. IEEE 1588 which makes the clocks in the system synchronous by message exchange with a preselected main clock can provide precision at the interval of 10-100μs for software implementation and below microsecond level for hardware implementation. [3],[4],[6],[7],[8]. Precision is decreased to some extent for the key based systems. IEEE 1588 has been fastly put into use in industrial systems and started to be implemented onboard on several PLCs.

Dependability is an important requirement for the applications that are being run on industrial control systems and have critical safety constraints [9]. Dependability concept also includes factors like availability, safety, integrity and maintainability. [10]. To call a distributed industrial control system that communicates over network dependable, dependability of both the network and the controllers should be ensured. When designing a dependable industrial communication network, dependable distributed synchronization and the consistency of the values that are sent with the messages with each other and with the system condition is important. The dependability problem is more prominent for RTE based solutions due to the Ethernet's nondeterministic properties [11]. Dependable communication should ensure that the right information is being sent to the right location at the right time and sequence. Dependability support is usually provided by allocation of static extra load according to the expected worst condition. [4]. As an example, for a TDMA based protocol, for resending of every lost message, allocation of extra time

segments as much as the time segments that are separated as nodes is required and only half of the capacity can be useful.

CHAPTER 3

RTXX PROTOCOL

When we look at the industrial networks, we basically see two approaches. First of these is the fieldbus [31] type networks that are stated in *Chapter 2*, the second which is also the subject of this thesis is the Ethernet based approach [32] . As also discussed in *Chapter 2* , fieldbus type approaches development costs are high, expansion and update of the existing system is both costly and complex. Another difficulty on this type of networks is increasing the data flow amount on the network. The Ethernet technology however is being spread everyday and its data flow capacity is continuously increasing.

When we look at the conventional Ethernet architecture, it can be seen that the shared medium access time interval is indeterminate. In other words, the access to the shared medium can be performed from several units simultaneously which can result in collision. Due to the collision avoidance algorithms that exist on the MAC layer of the Ethernet, collisions can be prevented but since it is not possible to estimate delay times because of these algorithms, usage in the industrial networks is difficult. To avoid this, Ethernet switch structures are being used today. This way, one physical medium is used for each connection which can handle the collision problem. But because of their costs, delays resulting from the Switch buffers and the QoS requirements, usability in the industrial networks is decreasing .

RTXX Protocol [30] offers a new approach on the Ethernet interface. In the first part of this approach, the time slot division of the physical Ethernet medium as real time and non real time according to the Time Domain Multiple Access (TDMA) principle is predicted. In the second part, providing the real time communication infrastructure

and ensuring the shared medium access to stay in the boundaries of the protocol rules is provided. With these properties, it can both offer the real time requirements of the industrial communications and the coexistence of the real time and non real time traffic in the same medium while providing the above mentioned advantages. In this thesis, the distributed separated event control approaches of the RTXX protocol in industrial applications will be discussed in detail, rather the communication model will be the main scope.

3.1 RTXX Protocol Architecture

RTXX is defined as a protocol to operate in distributed architectures [34]. According to this definition, each controller on the network, like PLCs, is defined as a *node*. Communication relationships between the nodes will be explained in the following sections. To be a brief description of the protocol, each node in the system knows when to send the next message and who the sender is. Thus, it organizes its functionality according to this information. The node which will send the data over the shared medium sends the message in a form of *communication request* in order to inform the other nodes in the system and make reservation for the future nodes which are waited a response from. The other nodes that receive the communication request, process the request and determine the next message sender on the network.

In order to avoid collision on the shared medium, RTXX Protocol proposes time-slotted access [30]. According to the protocol proposal, each time slot constructed as a fixed size time interval and the transmission of the messages should be done in the predetermined time slots. However, RTXX protocol proposes time-slotted access of the shared medium, it does not define the implementation method of time-slotted access over Ethernet bus. Within the scope of this study, implementation of time-slotted architecture over Ethernet bus will be explained in more details in *Chapter 3*.

When we look at the overall structure of the RTXX Protocol, we can analyze it in five sections;

1. Network Node

2. Message Structure
3. Communication Operation
4. Correct Network Operation
5. Non Real Time Traffic Support

Rather than explaining the discrete event control approaches of the RTXX Protocol, communication model of RTXX Protocol will be the main focused in this section. Certain terms and their definitions in RTXX protocol is explained in the following;

Shared events - Tasks: These are the events which organize the communication between system nodes and determine the operation of nodes.

Non-Shared Events: These are the events defined in the controller's communication models that control the internal operation of each node regardless of other system nodes.

Jobs: These are the communication messages which are transmitted during the execution of tasks. Many controllers may require communication between each other to perform a specific task. To establish this communications, jobs are used as a container on the shared medium.

For instance, If the operation of an air motor is a task in the system, each message transmitted over the system to operate the air motor is defined as jobs.

3.1.1 **Network Node**

Each network node R_i, in RTXX protocol has to implement the following entities;

- A communication model automaton CM_{Ri}
- An input buffer to store the input requests
- An output buffer to store outgoing messages
- An active task list currently initiated in the communication model
- A priority queue to store and sort the incoming request based on their deadline parameter

Communication model is based on the principle of the communication between distributed controllers on the same system that are interconnected by a shared medium. In this model, each controller is responsible to manage a subunit of the system and the lifecycle's of the controllers are determined by automata. Modeling principle and the execution of automata are illustrated in *Example 3.1*

Example 3.1: A small industrial bottle filling machine model is illustrated in *Figure 2* . The model consist of a conveyor belt (R_1), a filling unit (R_2) and a high level controller (R_3) which is responsible from the control of R_1 and R_2. The high level controller, R_3, can start the process (*sp*), inform the controllers that bottle is ready to fill (*rtf*) and the filling is finished (*ff*). R_3 is also responsible from the system security by preventing unwanted behavior of low level controllers as R_1 and R_2. The conveyor belt moves the bottles to the filling point and after the filling process it moves to bottles for further processes. Basic operation of the conveyor belt is, start the process (*sp*), run until bottle detected (*bd*) by the sensor and stop the belt (*sb*) after than wait until the *ff* and *sp* commands to start the whole process again. The filling unit waits until the *rtf* command, starts the filling (*sf*) process, finishes filling process (*ff*)and waits until the *rtf* signal to start filling process again.

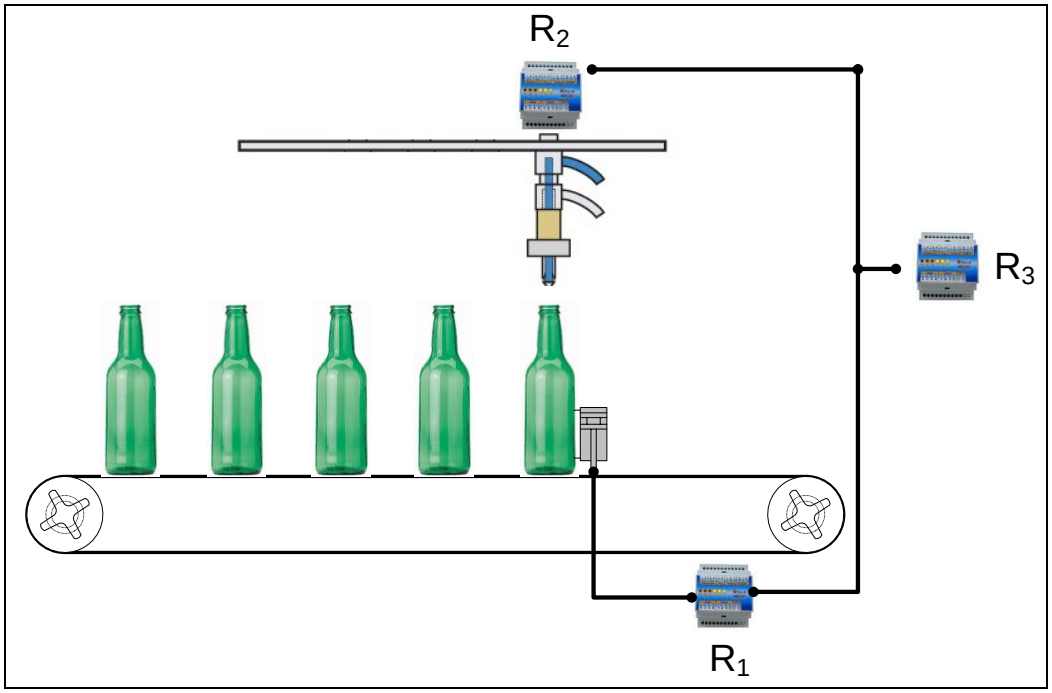

Figure 2: Bottle Filling Machine Model

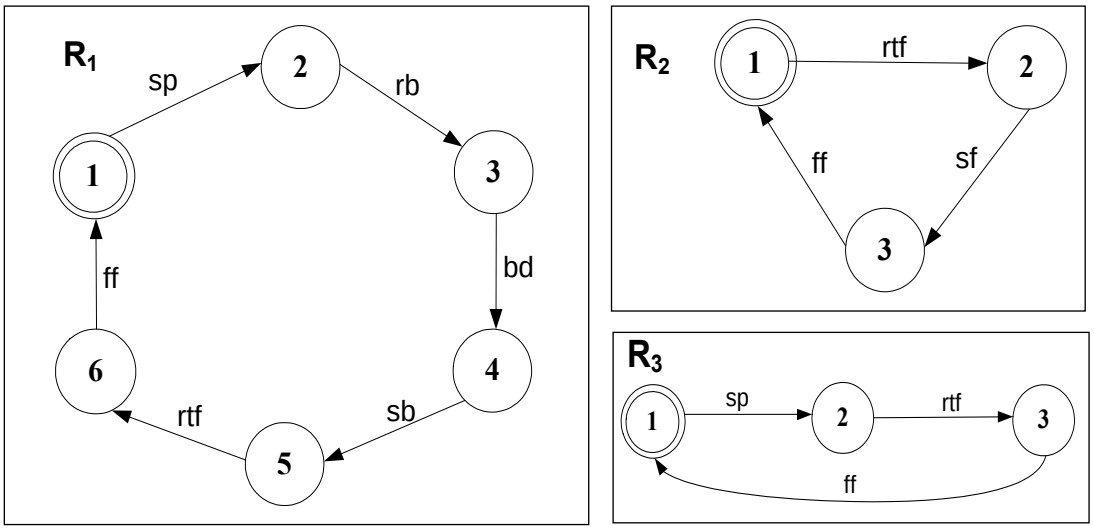

Figure 3: Discrete Event System Automata of Industrial Bottle Filling Machine

Each controller, R_i, has its own communication model, CM_{Ri}, in the system. *Figure 3* shows the discrete event system automata of the example system given in *Figure 2* and *Figure 4* illustrates the communication model of the example discrete event system. Every state in communication model CM_{Ri}, corresponds a state in controller R_i and the initial states are marked with a double circle. For instance, the states 1_1, 1_2, and 1_3 in CM_{R1} corresponds to state 1 in R_1. The communication between the controllers is expounded hereafter and it is assumed that each controller is at their initial states;

- The controller R_3 emits the question job $?sp_{R3}$ to deduce the R_1 is in initial state or not.

- The controller R_1 senses the question job $?sp_{R3}$ and it replies to R_3 with the $!sp_{R1}$ job when it is ready to start

- R_3 receives the $!sp_{R1}$ job and it sends the command job sp_c to make R_1 switch to state 1_3 to 2_1 which makes it to actuate the conveyer belt engine.

- To run the *rtf* task in the system, R_3 controller must know the states of the low level controllers R_1 and R_2. Therefore, R_3 sends $?rtf_{R3}$ question job to interrogate whether low level controllers are ready to execute this task.

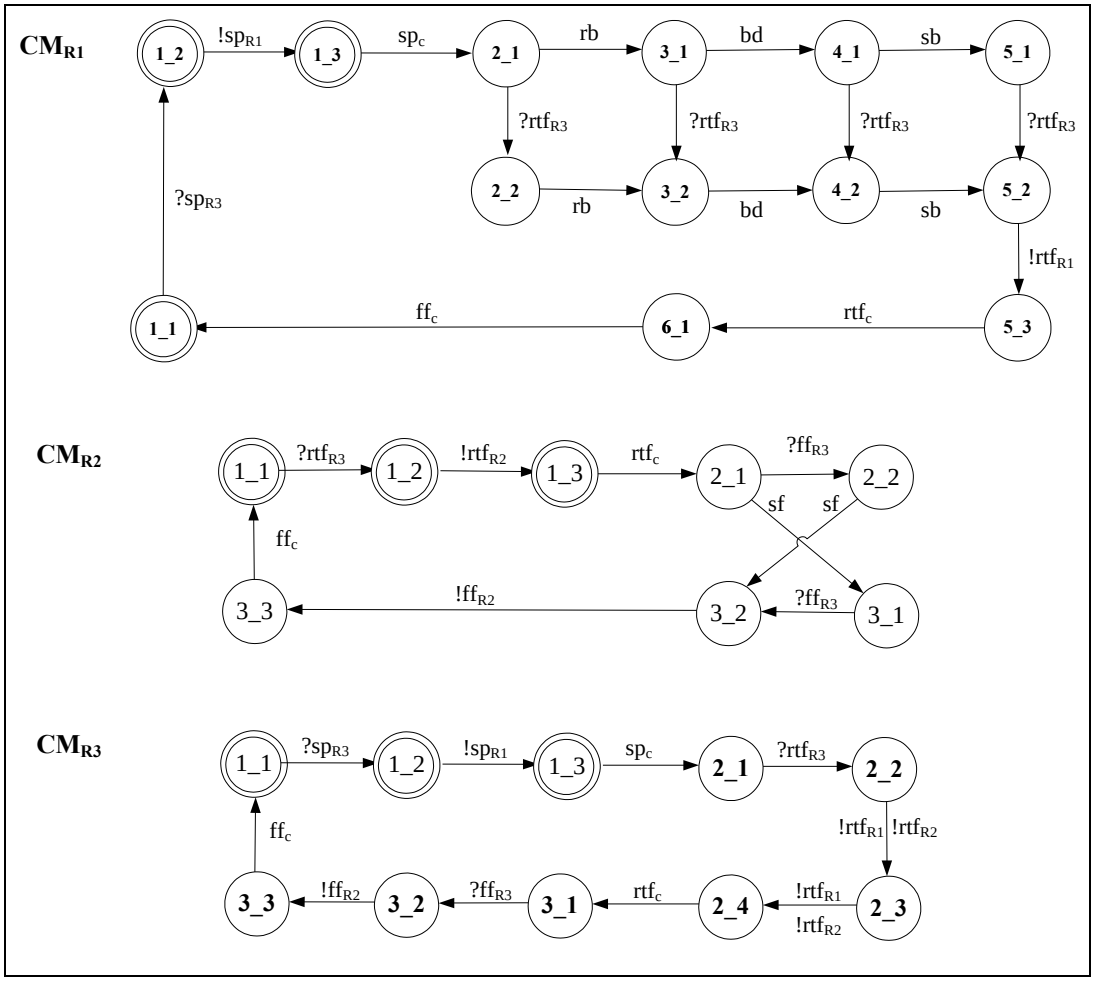

Figure 4: Communication Model of Bottle Filling Machine

- R_1 and R_2 receives the question job *?rtf*$_{R3}$.
- *?rtf*$_{R3}$ question job is applicable for R_2 at initial state thus it replies question job with the *!rtf*$_{R2}$ job immediately and state in the CM_{R2} switches from 1_2 to 1_3.
- On the other hand, *?rtf*$_{R3}$ question job is not possible until the R_1 finishes the non-shared events rb, bd and sb. After R_1 finishes the non-shared events, it replies the R_3 with the *!rtf*$_{R1}$ job and switches the state in CM_{R1} from 5_2 to 5_3.
- R_1 and R_2 respond to R_3 controller, R_3 emits *rtf*$_C$ command job to execute the *rtf* task in the system and switches the state in CM_{R3} from 2_4 to 3_1.

- With the rtf_C command job, CM_{R1} switches to state 5_3 to 6_1, CM_{R2} switches to state 1_3 to 2_1 and R_2 controller starts to filling process.

- Communication keeps going with the interrogation of the current situation in low level controllers for *ff* system task by R_3.

To exploit a system task, each job related to the task must be totally completed. If we denote the n*umb*er of jobs relate$_d$ to task σ by N_σ, then we can describe the deadline for each job as $d_j := r(\sigma) / N_\sigma$ where $r(\sigma)$ illustrates the time interval between the occurrence of the event and its execution. In accordance with this definition, we can say that the jobs that are ready to be sent must be transmitted at least in d_j time. As an illustration, If the job ready to be sent at t_0 instant, then it must be sent at t_0+d_j latest. By this definition, the communication model is combined with the deadline parameters that communication model is made into a real time communication model.

3.1.1.1 Priority Queue

Priority queue is used for storing the requests that are defined as a tuple in the system in form of (*N,e,d,T*) which indicate the transmission of the the future requests. *N* represents the Node which the request is relevant to. *e* represents the eligibility time that the demander Node requires to finish its internal process to accept the reply messages. *d* represents the deadline parameter of the job and *T* represents the active task that issued the request. Priority queue stores each request as it is and order them by their deadline and eligibility time elements.

3.1.2 Message Structure

In RTXX Protocol, each job in the system is transmitted as a message on the shared medium. The Protocol proposes to send only one message within a time slot whose interval must be determined by the longest size that a message can be constructed to prevent the shared medium from physical collisions.

The message structure of RTXX protocol consist of the following entities.

- A set of jobs to be sent by the node N_i.
- A set of receiver nodes.
- A minischedule contains the communication requests.
- A set of terminated tasks in node *Ni.*

In this study, in Chapter 5, mult*i-m*essage per slot approach is implemented to achieve timing overhead arising from the locking mechanisms in Linux operating system. Multiple accesses on shared medium are prevented by the priority queue implementation in *Section 5.1*.

3.1.3 Communication Operation

Priority queue constitutes the main foundation of RTXX Protocol. The following part explains the initial and subsequent runtime operation of the protocol.

- The highest level controller generates an initialization message and transmits it on the shared medium to other controllers in the system. At this time, each controller is on its initial state and monitors the physical transmission line for possible incoming messages.
- Controllers captures the request from the shared medium and stores it in their priority queue
- After the initialization, each controller takes out the nearest deadline eligible communication request from the priority queue.
- The node which take place in the request sends its message into output buffer and sends it through the shared medium

- Every node in the system inserts the communication requests in the minischedule to their priority queues. Thanks to this, each node has the same unique priority queue order that makes the system fully synchronized and make the nodes choose the same node as the sender.

- The receiver nodes in the system capture the incoming message, put it to their input buffer, update their state according to the computations, generates the outgoing message and put it into their output buffers

Protocol follows the same way for each real time message in the system that makes it work properly during the lifecycle of the system

3.1.4 Correct Networked System Operation

Each ready job in the system must meet the deadlines and are transmitted in accordance with the communication model to warranty correct system operation. Some network and transmission parameters must be defined and computed to provide real time characteristics and correct operation. Hereafter, definition and the calculation of these parameters will be illustrated.

The most basic requirement for the correct operation is computing the frequency of the real time slot and bandwidth of the real-time traffic. The calculation of minimum tolerable transmission frequency of real time slot is illustrated in *Equation 1* [30]. Q_{max} represents maximum estimated priority queue size, d_{min} represents the minimum job deadline and e_{max} represents the maximum eligibility time.

$$r_{min} = (Q_{max}+1) / (d_{min}-e_{max}) \qquad (1)$$

After determining the minimum tolerable transmission frequency r_{min}, with the knowledge of the largest message size that can be created on the system F_{max}, the minimum bandwidth requirement for real time traffic can be calculated as in *Equation 2*.

$$C \geq r_{min} \text{ x } F_{max} \qquad (2)$$

With the help of Equation 1 and, bandwidth and the frequency requirements can be calculated for the correct networked system operation.

3.1.5 Non-Real Time Traffic Support

If the total bandwidth in the network is considered as C, then the amount of the available bandwidth excluding RTXX Protocol can be calculated as $C - (r_{min} \text{ x } F_{max})$. Usage of this additional bandwidth for non real time applications is proposed by the RTXX Protocol as follows;

r_{min} value in *Equation 1* is defined as the minimum tolerable transmission frequency for real time slots. By this definition the period of the real time slots can be defined as $1 / r_{min}$. Protocol proposes to split this period into *G* time slices, named as time slots, assign the first time slice to real time traffic and the other *G*-1slices to non real time traffic. By this approach, the protocol both guaranties the real time communication bounds for real time traffic and opens the shared medium for non real time traffic usage.

Protocol proposes to schedule the non real time slots with a pre-computed transmission schedule. In each non real time slot, only one node can access the shared medium and transmit its messages according to transmission schedule. If there are no eligible request during a real time slot, protocol also proposes to use that

real time slot for non real time traffic. *Figure 5* illustrates the scheduling of real time and non real time slots for the period of 1 / r_{min} and *G*=5. As can be seen in the *Figure 5*, Slot 11 is used for non real time instead of real time traffic

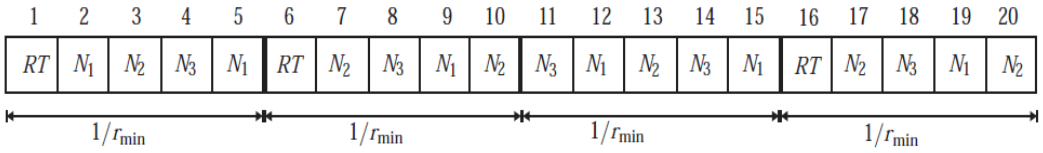

Figure 5: A Sample Transmission Schedule of RTXX Protocol [30]

CHAPTER 4

REAL TIME ETHERNET MEDIUM ACCESS IMPLEMENTATION

As we discussed in previous sections, communicating over shared Ethernet networks is nondeterministic as more than two nodes can start to transmit packets at the same time. In a collision situation, collision avoidance algorithms in MAC layer of IEEE 802.3 [12] tries to retransmit buffered data after random intervals which make delay bounds unpredictable. To avoid this kind of collision sources, this study proposes Time Domain Multiple Access (TDMA) approach in the Ethernet level for Linux operating system.

4.1 Time Domain Multiple Access

TDMAController() thread constitutes the core Real Time Ethernet implementation. Simply, it divides Ethernet Bus into pieces, called time slots, over temporal plane and coordinates the bus access between applications and Linux kernel. These time slots can be dedicated to real time protocols like RTXX, non real time traffic like IP, IEEE 1588 Protocol or another custom protocol on demand. With this flexibility, bandwidth allocated to protocol applications may be configured statically or dynamically.

The main objective of TDMAController() thread is managing the schedule of the packets to be sent over Ethernet Bus. TDMAController() consists of a periodic timer and locking mechanism for related protocol threads.

4.1.1 Timing Mechanism

Two of the most important timing requirements for TDMA Controller implementation are, high precision and high accuracy periodic timers and a real time capable operating system kernel.

4.1.1.1 Timer Clock Accuracy

The accuracy and the precision of the timer counter clock in the system determine the reliability of time interval of each time slot. In order to provide deterministic and predictable slot intervals, timer clock source should be selected as accurate as possible. This accuracy is dependent on hardware clock source and driver interface of this clock hardware.

4.1.1.1.1 Hardware Clock Sources

Building a scheduler, which is the core of an Operating System, requires keeping track of time to switch between parallel executed software threads. There are several timer devices produced for this aim. Common architecture for this timer devices usually covers an oscillator and a counter components. When the oscillator provides input frequency for the timer device, the counter controls counting process for each cycle of the oscillator. The counter is controlled by software which defines counter inputs, modes (one-shot or periodic) and generates a state signal which may interrupt the processor at any time.

PC Timer Devices and Components are considered below [13].

The Programmable Interval Timer (PIT)

The oldest PC timer device is the PIT. PIT has a 1.193182MHz input oscillator, 16-bit counter and counter input registers. The PIT covers three timers to manage system timekeeping. Timer 0 provides interrupt signals, Timer 1 refreshes RAM and Timer 2 generates tone signals for PC speaker. This timer was not suitable for timekeeping of today PCs, because it was designed for the first PC architecture.

The CMOS Real Time Clock (CMOS RTC)

The CMOS RTC is a function of the battery-backed memory device. There are two functions in the RTC. One of them keeping the time of day (TOD) information and the other one is generating periodic interrupts by a timer clock. But it is not possible to read or write on the counter. Only counter inputs can be set to any power of two rating from 2Hz to 8192Hz.

The Local Advanced Programmable Interrupt Controller (Local APIC)

The Local APIC is another timer device in multiprocessor systems. Each processor has one local APIC. The Local APIC has a 32 bit counter and a number of counter input registers. The input frequency depends on memory bus frequency before the multiplication. So, counter capability is wider than PIT or CMOS. But counter frequency cannot be determined by software.

The Advanced Configuration and Power Interface (ACPI)

The another additional system timer is the ACPI which is known the power management timer. ACPI has 24 bit counter but does not have a counter input register. The ACPI timer is used to control power saving functionality.

Time Stamp Counter (TSC)

The TSC is a 64-bit cycle counter which is used on new type processors. It has no capability to generate interrupts and does not have counter input registers. TSC can be read by software with the rdtsc instruction. But the rdtsc instruction can be used in user mode and there would be an issue on operating system side. Also there are a few drawbacks for TSC such as:

- There is no reliable way to determine the TSC's input frequency
- There are several power management technologies which change Processor's clock speed dynamically with little or no notice.
- The TSC can be stopped in their lower-power halt states

HPET (High Precision Event Timer)

The HPET device is available in new generation PCs. It has 32 or 64 bit counter which runs until stopped by software. The HPET has multiple timers which have timeout registers that are compared with the central counter and takes measures against the timeout if the timer was set to be periodic. Thus, The HPET is used instead of the PIT and the CMOS periodic timers. [14], [13]

4.1.1.1.2 Timer Interfaces provided by Linux Kernel

Standard Linux distributions only implement system clock timers, called *jiffies*, that has the maximum frequency of 1000Hz with an unpredictable accuracy and alarm mechanisms based on RTC clock in the BIOS which has the maximum timer period of 125uS for software developers. Other clock sources are dedicated to specific procedures like power saving mechanisms and hardware synchronizations. The Real Time Ethernet implementation requires accurate and high resolution user timers, so standard Linux distributions can not be used for this implementation.

First prototypes of high resolution timer support for Linux kernel were developed by Thomas Gleixner for Intel x86 architecture. During the time, there were several updates and new kernel patches were developed and merged in to Linux kernel by Thomas Gleixner [15] and Ingo Molnar. By this patch, POSIX [16] timers became as accurate as the hardware clock allows which is about 1 nanosecond HPET timer today.

There are several types of POSIX high resolution timer sources in the real time kernel which can be selected during the timer creation routines as CLOCK_REALTIME, CLOCK_MONOTONIC, CLOCK_PROCESS_CPUTIME_ID and CLOCK_THREAD_CPUTIME_ID. All of these timers use the same high precision clock hardware to generate timer interrupts and signals. CLOCK_REALTIME timer mechanism uses system-wide high resolution timer for timer expiration interrupts and signals. Timer period for this type of timer may change if someone changes the system timer during the lifecycle of the

working thread. CLOCK_MONOTONIC timer uses high precision timer which starts to count from 0 when the system powered up and can not be set/reset until the next power on. Timer period for this type of timer is not affected by changes in system timer so this timer type is suitable for most of the periodic applications. CLOCK_PROCESS_CPUTIME_ID and CLOCK_THREAD_CPUTIME_ID can be used to measure the consumed CPU time for applications [17].

In our TDMA Controller implementation CLOCK_MONOTONIC type of timer is selected to generate periodic and stable timer events.

4.1.1.1.3 Signaling and Real Time Scheduling Characteristic

TDMA Controller implementation uses signals in the operating system to trigger the slot switching mechanism resulting from timer interrupts. Therefore the signaling performance and real time scheduling characteristic of the operating system is the third major factor in the Time Slotted Architecture after timer clock hardware and timer clock interface.

Standard Linux signal delivery times and thread switching times may vary depending on CPU load of the system. For this reason real time characteristics of signaling and thread scheduling should be implemented to Linux kernel. In order to provide this capability to Linux Kernel, PREEEMPT_RT [18] kernel patch is applied to system [19].

Signals in Linux operating system are used to notify a thread or process of a particular event. Signals may also be called as *software interrupts*. Standard Linux kernel implements 31 signals for interrupt and exception handling, synchronization and inter process communications (IPC) in the system. POSIX standard adds a new class of signals called *real time signals* ranging from 32 to 63. There are some major differences between normal signals and POSIX real time signals. The first difference is the priority level of signals. For normal signals, lower numbered signals have higher priority then higher number signals. In contrast to normal signals, POSIX real time signal priorities proportional to signal numbers. The second difference is the

signal queuing mechanism. POSIX real time signals uses signal queues to guaranty successfully delivery of signals to the related application. On the other hand normal signals is merged if a process has already a have pending signal, so only signal is delivered for overall signal activity [20].

In TDMA Controller implementation highest priority POSIX signal, SIGRTMAX, is used to handle timer interrupts. In this way, in case of a signaling latency, TDMA Controller application always know the time slot it is related to and will not miss any timer interrupt during its life cycle.

In addition to high resolution timer support implementations, Ingo Molnar developed real time preemption kernel patch which is providing fully priority preemption mechanism to the Linux kernel. By applying preemption kernel patch and high resolution timer support, standard Linux kernel gains real time capabilities which makes it suitable for Real Time Ethernet implementation.

For this study, Linux kernel version 2.6.32.12 is selected and related real time patch is used to build a real time capable Linux kernel. Configuration and the build process is explained in *Appendix B*.

4.1.2 Locking Mechanism

Locking mechanism is the other main requirement for Real Time Ethernet implementation. Main idea behind the locking mechanism is to prevent simultaneous access to physical Ethernet layer. In order to provide this capability to TDMA Controller thread, POSIX mutexes, conditional variables, semaphores and user defined system calls are used by TDMA Controller application.

In Real Time Ethernet implementation, to ensure the system's proper operability it is required that all the nodes in the system work in accordance with the Real Time Ethernet implementation. Even though the implementation supports real time and non real time communication, if a standard Ethernet connection is realized over the shared medium, the traffic flow over the shared medium may be disrupted by the

packets that will transferred from this node . In this study, it is assumed that all the nodes in the system support Real Time Ethernet protocol and the access of the nodes to the medium is performed inside the predetermined time slots. In future work, the possible actions for the error cases resulting from the reasons mentioned above may be discussed.

It can be observed that the Real Time Ethernet implementation is split into real time and non real time slots in the shared medium. In these two types of slots the behavior of the Real Time Ethernet implementation differs. When the system is in the real time slot, non real time traffic flow is blocked in every node that forms the system. As the blocking mechanism, the blocking APIs which the driver functions use for warning the system when the hardware transmission buffers are full are called. With this method, all the non real time traffic in the non real time slot interval can be stopped while the system's general behavior are not affected. The non real time traffic that is stopped, like the video streaming or FTP, is buffered by the buffering mechanisms during the blockage interval. So, any corruption or packet loss is prevented for the Layer 3 protocol connections.

The base effect of the blockage to the system is the inability to deliver the Layer 2 and Layer 3 protocol packets during the real time slot interval and buffering delays for these unsent packets. If the frequency and the interval length of the real time slots are increased, the buffering amount on the networks stack starts to increase according to the non real time traffic density on the node. The buffering structure on the network stack is dynamic and limited with the memory amount on the node. If the memory amount on the node starts to be insufficient, buffering is stopped and packet loss occurs. Moreover, the high waiting time for the packets may also result in Layer 3 protocol connection cut offs. When the system is in the non real time slot, real time traffic is blocked and non real time traffic is allowed to flow. The structure of the real time traffic and packet delay amounts can change according to the type and execution manner of the protocol running in the real time slot.

Filling the network stack buffers will take time because of the high memory amount when the experiment setup is considered. So this experiment is not performed. Instead NRT delay experiments that measure the delay time of real time traffic is performed.

The aim of this experiment is to detect the delays occurring on the non real time traffic depending on the density of the real time traffic.

First part of the locking mechanism is POSIX based synchronizations objects. There are three types of POSIX objects used in TDMA Controller implementation which are mutexes, conditional variables and semaphores.

4.1.2.1 Mutexes

Mutexes are the synchronization mechanism aims to prevent race conditions over shared objects and generally used by driver functions. For Real Time Ethernet Implementation, TDMA Controller must lock and release threads periodically based on slot type information. Because of this TDMA controller creates one POSIX mutex for each thread that has possibility to access Ethernet layer. Basic usage of the POSIX mutexes illustrated in *Code 1*

```
/* Function C */
void foo()
{
  pthread_mutex_lock( &mutex );
  testcount--;
  pthread_mutex_unlock( &mutex );
}
```

Code 1: Sample Usage of POSIX Mutexes

4.1.2.2 Conditional Variables

Condition variables are the synchronization mechanisms that block the waiting task until a specific condition is true. Conditional variables must be used with a mutex variable to avoid race conditions. If a thread signals another thread with a condition, the signaled thread will be unlocked if the condition is true, otherwise signaled thread will locked again. In TDMA Controller implementation, every element that can access the network transmission in a time slot is controlled by these conditional variables. A simple locking mechanism for conditional variables is illustrated in *Code 2*

```
Thread1()
{
    pthread_mutex_lock( &mutex );
    while( testCount < 10  )
    {
      pthread_cond_wait( & cond, &mutex );
    }
    pthread_mutex_unlock( &mutex );
}
Thread2()
{
    pthread_mutex_lock( &mutex );
    testCount--;
    pthread_mutex_unlock( &mutex );

}
```

Code 2: Sample Usage of POSIX Conditional Variables

4.1.2.3 Semaphores

Semaphores are the third synchronization mechanism used by the TDMA Controller implementation. By definition, semaphores are kernel objects that contain a variable which can be checked and modified by the processes and threads. Semaphores are one of the fastest synchronization mechanisms in Linux operating system. There are two types of POSIX semaphores. First one is the named semaphore which can be accessible between separate processes, the other one is unnamed semaphore which is accessible only in process memory which threads in the process can be use to synchronize between each other. TDMA Controller implementation uses unnamed semaphore to prevent access from other application processes. A simple synchronization mechanism for POSIX semaphores illustrated in *Code 3*

```
timerHandler(){
        sem_post(&timer_sem);
}
periodicThread(){
        for(;;){
                sem_wait(&timer_sem);
                do something;
        }
}
```

Code 3: Sample Usage of POSIX Semaphores

Standard Linux system, memory space is separated into two distinct regions as user space and kernel space. Linux kernel and all of the kernel services including device drivers run in the kernel space and applications run on the user space. Accessing distinct regions is prohibited by Linux kernel. In order to provide kernel level services to user applications like capturing data from keyboard, mouse or a webcam, Linux kernel offers special gateways to user space programs named system calls.

4.1.2.4 System Calls

System calls are the services provided by Linux kernel to attract kernel resource from a user space application [21].

A system call is executed in the kernel space and a user program is executed in the user space. In a Linux system, hardware access is restricted to the kernel space to protect the hardware routines from user space programs. Some cases, user space application requires to access directly to hardware to perform the specific job, like high precision timer access. In this case, there is a special need for bridging user programs to hardware which is called “system calls”.

System call implementations are dependent to microprocessor architecture. Every system call has a unique number associated with it. For Intel x86 architecture, when a user space program calls a system call, a library routine traps the kernel via executing the special “INT 0x80” assembly instruction and the associated number of the system call is passed to kernel via EAX register.

The arguments of the system call are also passed to kernel via EBX,EBC, etc. register. Return value of the system call is passed from kernel to user program via other CPU registers. [22]

Detailed information about implementing system calls to Linux kernel is explained in *Appendix A* In this implementation, TDMA controller needs to control and access Ethernet hardware which is located in kernel space. Standard kernel does not offer system calls to access Ethernet hardware from user space so new system calls are required for TDMA Conroller.

There are three new user defined system calls added to Linux kernel which are lockNetDevice(), unlockNetDevice() and sendNetDriver(). lockNetDevice() system call locks the upper layer packet flow to Ethernet physical layer. When a real time traffic dedicated slot is activated by TDMA Controller, this system call used for locking Linux network stack. unlockNetDevice() system call releases the Linux network stack. When a non real time dedicated slot activated by TDMA controller,

this system call used for unlocking Linux network stack. sendNetDriver() system call sends a packet directly to Ethernet driver routines. When the upper layer traffic is suspended by the lockNetDevice() system call, real time protocol applications can not use standard socket APIs to access Ethernet hardware. In order to bypass Linux network stack, real time applications must use this system call to send their packets to Ethernet hardware. This system call involves some changes in driver level to add additional access capability to driver functions.

Standard Linux network device drivers must have some specific procedures to interact with Linux kernel. This procedures are defined in the “net_device” structure and device drivers should be written based on this procedures. Linux network stack uses a specific procedure reference pointer, called “hard_start_xmit”, to send buffered packet to physical layer and Linux kernel provides some locking mechanisms on this procedure to prevent multiple accesses simultaneously.

In the network device driver source code, there is a hardware transmit function which is referenced by this “hard_start_xmit” procedure. Linux network stack uses that reference to send next packet in the network stack queue to network hardware buffers. In order to send network packets to Ethernet hardware without using standard operating system interfaces, there should be a backdoor in driver source code. In order to provide minimum impact on device driver source code a new transmission function, ”psudo_start_xmit” , is added to source code as a backdoor instead of changing standard driver functions and “hard_start_xmit” routine forwarded to that psudo transmission function instead of real transmission function. Psudocode for pseudo transmission function illustrated in *Code 4*.

```
psudoTransmitFunc () {
        if (timeSlot==REALTIME){
                lockNetwork();
        }
        else{
                if (nonRtTrafficType==ONESHOT){
                realTransmitFunc();
                lockNetwork();
                }
                else {
                        realTransmitFunc();
                }
        }
}
```

Code 4: Psudocode for Psudo Transmission Function

4.1.3 Synchronization Mechanism

Synchronization mechanism is the third main requirement for Real Time Ethernet infrastructure. All the nodes in the Real Time Ethernet network must synchronize their clocks and periodic timers to the master node to communicate between each other.

First step of the synchronization is the system clock synchronization.

4.1.3.1 System Clock Synchronization

System clock synchronization algorithm is determined as the IEEE 1588 Precision Time Protocol [23] for this implementation.

4.1.3.1.1 IEEE 1588 Precision Time Protocol

Measurement and Control Systems are used in many manufacturing technology and many other areas of industrial automation. Day by day, timing requirements are

growing rapidly and new time synchronization methodologies are needed by the industry. IEEE 1588 PTP is a protocol that mainly targets the timing requirements coming from this necessity.

The first protocol studies about Precision Time Protocol were done in the 1990s by Agilent Technologies [24]. Then to meet the industrial requirements, academic and industrial studies for time synchronization is re-established in 2000s and following that studies, first version of the standard was published in November 2002 by The Institute of Electrical and Electronics Engineers, Inc. (IEEE)[25] as "IEEE Standard for a Precision Clock Synchronization Protocol for Networked Measurement and Control Systems" [23]

IEEE 1588 is a high precision time synchronization protocol suited for network and industrial systems which defines time synchronization methodology over packet based networks, like Ethernet.

IEEE 1588 PTP differentiates from older synchronization methods used in measurement and control systems by the method. Older control and communication standards uses a dedicated synchronous connection to capture frequency from the control link and requires specific hardware modifications which makes them to follow strict hierarchy. However IEEE 1588 PTP calculates the time by a specific PTP algorithm and determines the frequency by this calculations. The time distribution hierarchy determined by the clock source quality in spite of spatial location that makes it more flexible then older standards.

4.1.3.1.2 IEEE 1588 PTP Network Configuration

IEEE 1588 PTP Network consists of Master and Slave nodes in the system and creates a hierarchy based on this network elements. *Figure 6* illustrates a sample IEEE 1588 network topology. IEEE 1588 network is a spanning tree network which a Slave node in a level can be a Master node for sublevels of the tree.

The Master node on the top layer of the spanning tree uses a primary clock source, named as Grandmaster Clock, which may be a GPS, Rubidium or a trusted clock to synchronize Slave nodes in the network.

In the medium levels of the tree, a simple node acts like a bridge between upper layers and lower layers and uses Boundary Clocks to synchronies lower layer Slaves nodes based on Grandmaster clock. Generally medium layer nodes are physical network switches have support for IEEE 1588 Precision Time Protocol in addition to standard switching capabilities. Bottom layer nodes synchronize their clocks, named as Ordinary Clock, based on upper layer Boundary Clocks.

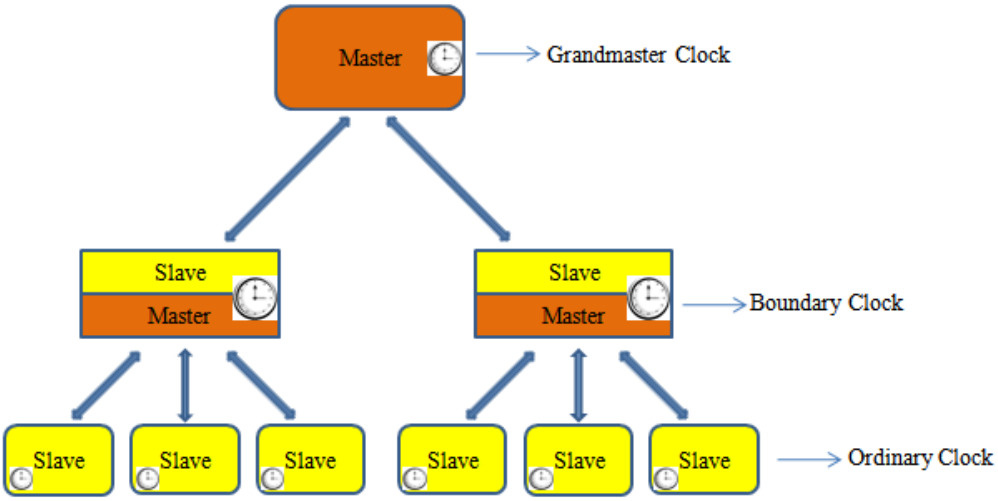

Figure 6: IEEE 1588 PTP Network Topology

4.1.3.1.3 Synchronization principle of IEEE 1588

CSMA/CD procedure in MAC layer of the IEEE 802.3 [12] Ethernet interface may cause to time packages being delayed or disappearing completely. IEEE 1588 PTP protocol offers a special time synchronization method to achieve this nondeterministic behavior.

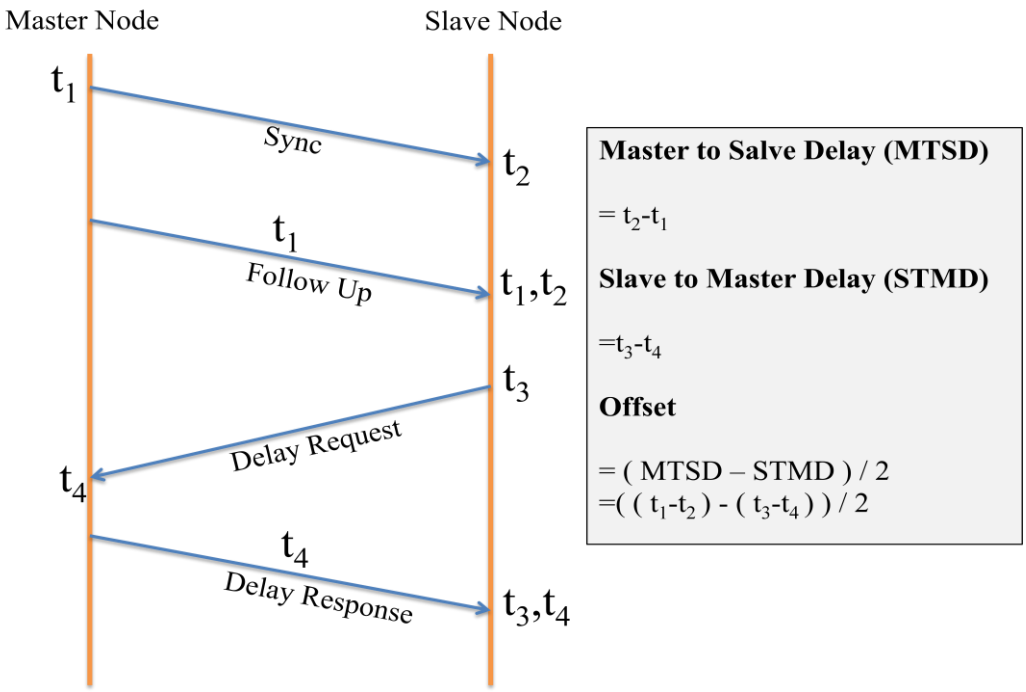

Figure 7: IEEE 1588 Synchronization Mechanism

IEEE 1588 uses relatively simple procedure for calculating the clock offset in the network which is illustrated in *Figure 7*

In the *Figure 7* two vertical lines indicates the time line for both master and slave devices.

- In the first step, Master node sends a sync message to the Slave node. Master node takes timestamps (t_1) when the message leaves the node.
- Slave node records the timestamp (t_2) when it receives that message.
- The master node then sends a follow up message to the slave node which is carrying payload of the original timestamp (t_1).
- At that time, slave node has both timestamp 1 (t_1) and timestamp 2 (t_2) and calculates Master to Slave Delay (MTSD).
- Then Slave node sends a delay request message and timestamps this (t_3).

- Master node timestamps the reception of this message (t_4) and sends a delay response to the slave which is carrying the payload of the timestamp (t_4) of delay request.
- Slave receives the delay response. After that time, Slave node has the timestamps to calculate the Slave to Master Delay (STMD) and the total offset between the nodes. After the calculation Slave node synchronies its clock to Master node.

Time stamping accuracy is the main factor affecting the success of the IEEE 1588 synchronization which should be made as close as the physical layer.

In addition to clock synchronization, clock rates should be adjusted with the protocol due to clock frequency differences in the system. Rate correction is done by measuring subsequent synchronization cycles and calculating differences between start of packets in Master node and arrival of packets to Slave node. Rate correction accuracy is dependent to synchronization period that IEEE 1588 PTP uses [26]. The more frequent the synchronization is, the more accurate is the rate correction. Rate correction algorithm is illustrated in *Figure 8.*

There are several implementation methods for IEEE 1588 in the literature. Basically implementations can be separated two groups as;

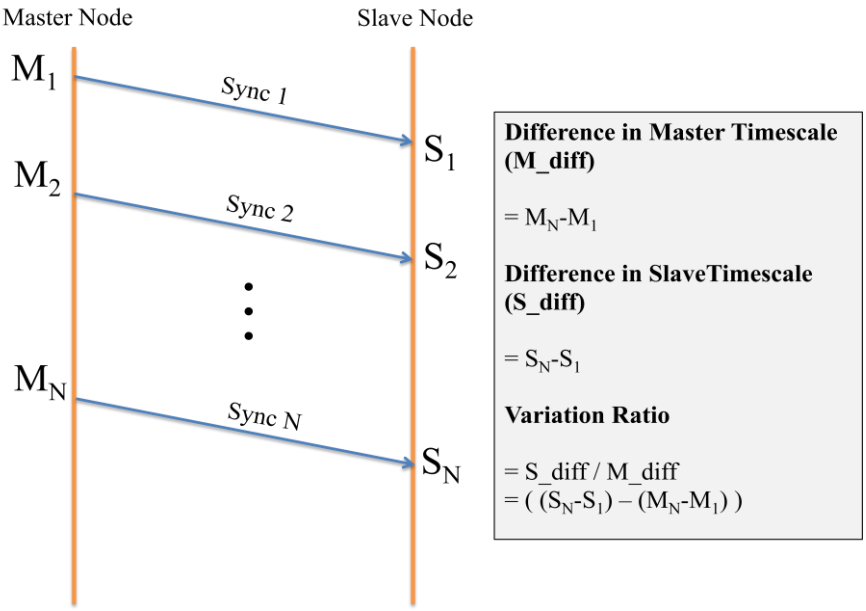

Figure 8: IEEE 1588 Clock Rate Correction Mechanism

4.1.3.1.4 Software Only Implementation:

This implementation is the basic implementation for IEEE 1588. All of the synchronization mechanism implemented via software and it does not requires any hardware assist which makes it very flexible to adapt newer platforms. However the software complexity makes the design harder and timestamping performance suffers from software based delays and jitters. Because of this delays, software based implementation is the least precision solution and results >10uS synchronization accuracy. This kind of accuracy is enough for process based communication requirements like factory automation.[27]

4.1.3.1.5 Hardware Assisted Implantation:

This implementation requires specific hardware modifications on network interface modules. There are several hardware implementations on FPGA, ASIC, Microcontroller or Ethernet controller based architectures. In terms of performance hardware assisted solutions has much higher accuracy than software based solution which is less than 10 nanoseconds today. However hardware assisted solutions are

much expensive and strict to specific hardware. Table 1 illustrates the comparison between hardware and software based solutions in terms of development considerations and precision performance [27].

Approach	Development Considerations	Precision Performance
Software Only	Software development required	Precision is low for most applications Typical: >10 microsecond on single link
Hardware Assist in FPGA	Significant hardware change required Software and FPGA IP development required	FPGA approach timestamps at the Ethernet MAC level Typical: >30 nanosecond on single link
Hardware Assist in Microcontroller	Requires change to new microcontroller Existing software changes may need to be customized	Microcontroller approach timestamps at the Ethernet MAC level Typical: >30 nanosecond on single link
Hardware Assist in Ethernet PHY	Simple Hardware implementation	Tightest time synchronization Typical: <10 nanosecond

Table 1: Comparison of IEEE 1588 Implementations [27]:

In this thesis, implementation method is determined as software only implementation which provides flexibility and adaptability in comparison to previously done academic studies. During the implementation, the same hardware architecture used for Master Node and Slave Node in the system. Since the clock sources are the same, rate correction algorithm is not necessary for this implementation so rate correction algorithms did not included in the IEEE 1588 PTP implementation. Software Implementation architecture is illustrated in *Figure 9*

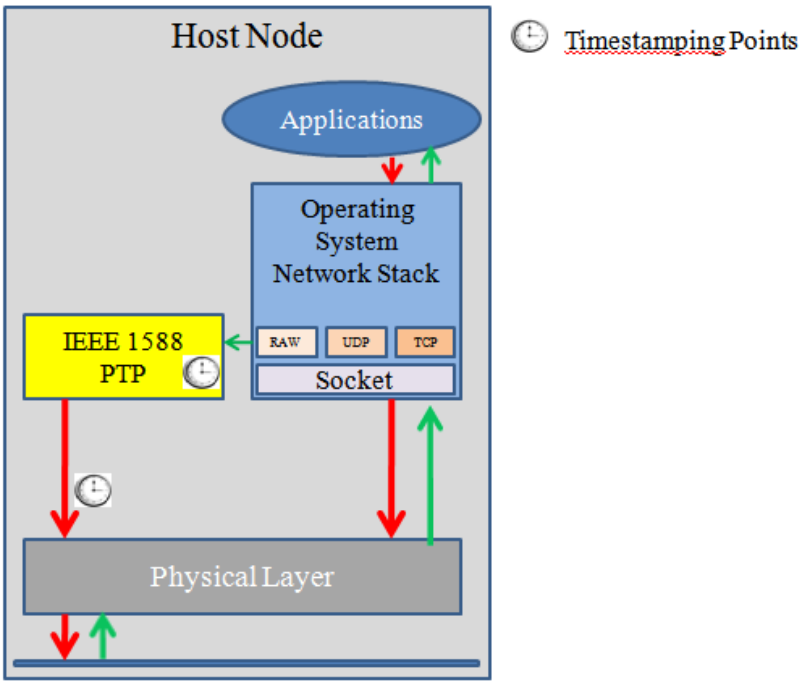

Figure 9: IEEE 1588 PTP Software Implementation

4.1.3.2 Periodic Timer Synchronization

Second step of the time synchronization is the periodic timer synchronization. Synchronizing system time is not enough for Real Time Ethernet implementation by itself, because nodes in the network must switch between slots at the same time as well. In order to provide this ability to nodes, additional synchronization over periodic timers is required.

System time is stored in the system as *timespec* structure. *timespec* structure consists of two long integers, each of two is 32 bit long. First long integer named as *tv_sec* which stores the seconds part of the system time and second long integer named as *tv_nsec* which stores the nanoseconds part of the system time. *Code 5* illustrates the timespec structure.

```
struct timespec {
long ts_sec; /* seconds */
long ts_nsec; /* nanoseconds */
};
```

Code 5: *timespec* Structure

A new synchronization algorithm has been developed to ensure the synchronization over periodic timers which is illustrated in *Figure 10*. Main idea behind the algorithm is capturing system time, t_s, and starting the periodic timer at "$t_s + t_\Delta$" . Value of t_Δ may change due to expected synchronization accuracy and the priority of the synchronization thread.

The synchronization thread enters a loop and compares the actual time , t_a , with the predetermined periodic timer start time, t_{start}, which is determined as the beginning of "t_s+2" second. If the actual time is greater than the t_{start} and "$t_a - t_{start}$" value is smaller than the predefined accuracy value, a , then related thread in the Node starts its periodic timer. Otherwise it increases t_{start} value by a predefined repeat period, rp , and keeps trying again and again until predefined accuracy a is achived. Synchronization algoritm may also be canceled after a specific number of tries if required. By this way, all the nodes in the system has the same system time as accurate as possible by IEEE 1588 PTP synchronization and has the same slot switching times with a worst-case drift of a.

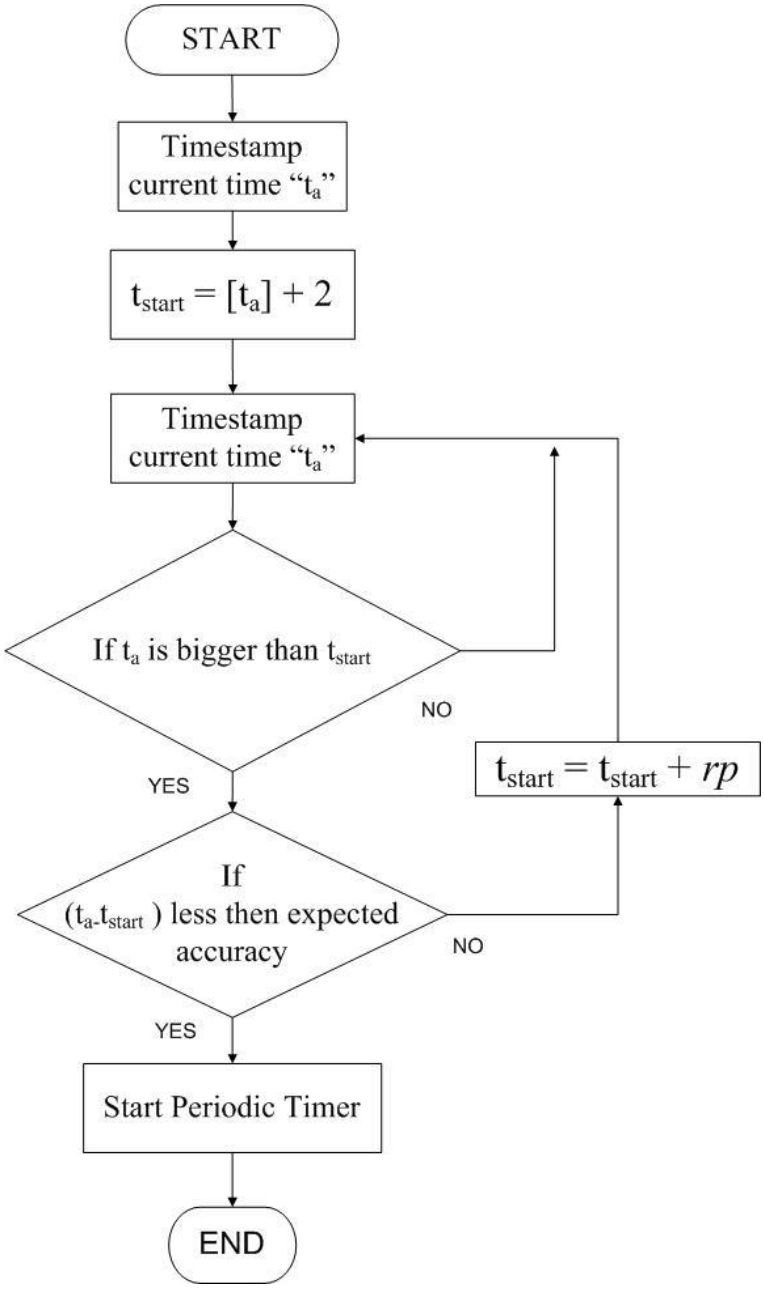

Figure 10: Periodic Timer Synchronization Algorithm

4.2 Programming Architecture for Real Time Ethernet Implementation

Real Time Ethernet Implementation contains four different functions to provide protocol functionality.

4.2.1 TDMAController()

The first and most important function is “TDMAController()” which is responsible to manage all the functionality including timing, locking and synchronization of Real Time Ethernet. This function is created as the highest priority thread in the operating system (priority = 80). Algorithm for this thread is illustrated in *Figure 11*

In addition to TDMAController() thread function, InitTDMAController(), IEEE1588Master() and IEEE1588Slave() also written to work with TDMAController() thread.

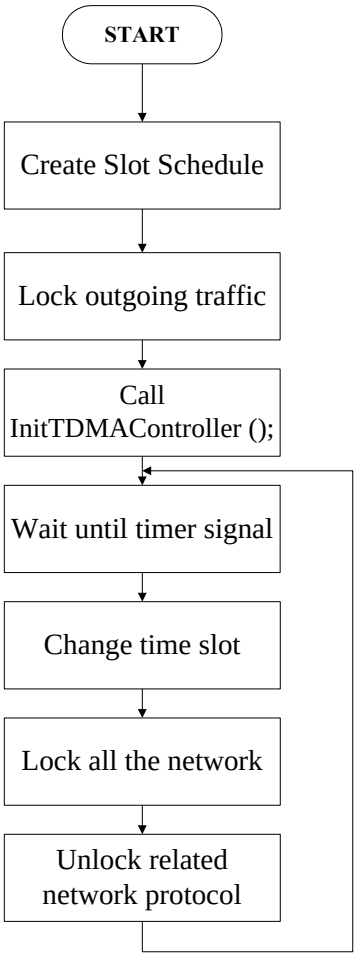

Figure 11: TDMAController() Algorithm

4.2.2 InitTDMAController()

InıtTMDAController() function is responsible to prepare all of the necessary conditions including starting IEEE1588 synchronization and periodic timer synchronization that TDMAController() needs. InitTDMAController() algorithm is illustrated in *Figure 12*.

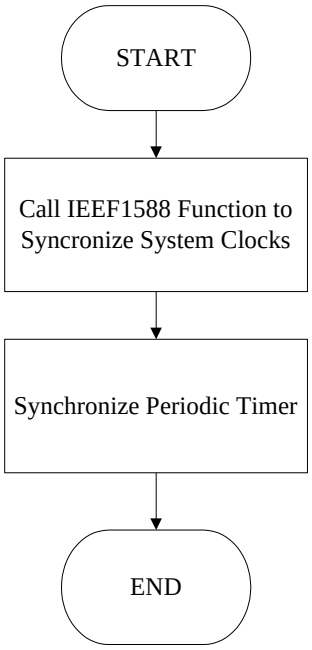

Figure 12: InitTDMAController() Algorithm

4.2.3 IEEE1588Master() and IEEE1588Slave()

TDMAController() functions changes its behavior based on operation mode. If the system will act as a Master node, the TDMAController() thread controls the IEEE1588Master() thread to synchronize Slave nodes in the system. Otherwise, TDMAController() uses IEEE1588Slave() to synchronize system time to master node in network. Algorithms for IEEE1588Master() and IEEE1588Slave() illustrated in *Figure 13* and *Figure 14*.

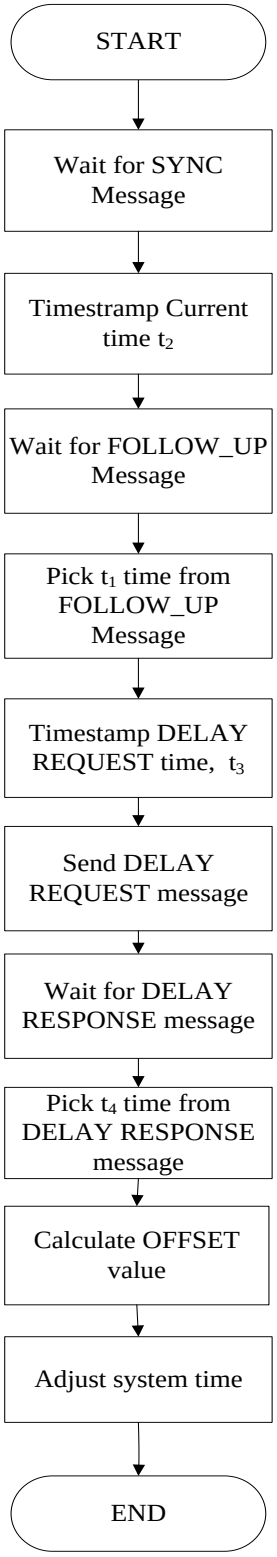

Figure 14: IEEE1588Slave() Algorithm

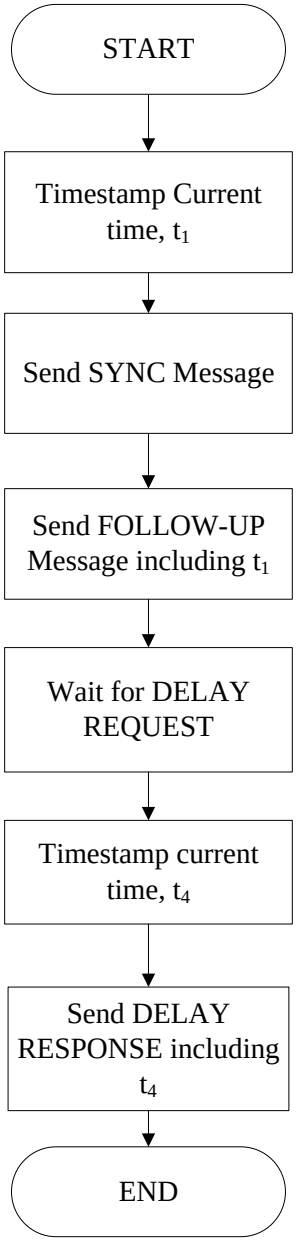

Figure 13: IEEE1588Master() Algorithm

CHAPTER 5

RTXX PROTOCOL IMPLEMENTATION

RTXX Communication Model implements message based Communication Requests which contains the following tuple to reserve communication channel for transmission.

Communication request tuple (*N,e,d,T*) consist of;

- A *NodeId* (*N*); a unique identifier representing the receiver node,
- an *eligibility time* (*e*), containing the time, when the receiver node is able to transmit a message,
- a *deadline* (*d*), based on protocol applicaition,
- a *TaskId* (*T*), that identifies the task, the message corresponds to.

Reservation of the communication channel is made by deadline parameter in the message structure which is sorted in a priority queue. Activation time of the reservations is determined by the eligibility time parameter in the message structure. If the eligibility time parameter of the communication request is smaller than the current time, request has processed by the priority queue and sorted by the system by deadline parameter otherwise communication request waits until eligibility time to be valid. A single node may request to transmit more than one communication request in a single message. In order to add this capability to RTXX protocol, communication requests are combined and named as *minischedule* in the message structure. Priority queues located in the nodes should be updated based on this minischedule during the lifecycle of the application. Because the communication medium is visible and accessible to all nodes, each node updates its priority queue

and contains the same scheduling table based on the minischedule of the transmitted message.

5.1 Processing and Priority Queue Management

RTXX Communication model requires a priority queue implementation to schedule outgoing packets in RTXX network. Scheduling of the packets changes dynamically based on the content of flowing traffic, eligibility time of precaptured packets and terminated task lists defined in RTXX Protocol.

This implementation assumes eligibility time of the packets and terminated task list member number is zero which means that packet is ready to be sent when it is created and there is no need to remove terminated task from priority queue. In this study, one global priority queue is created for each node to organize packet transmission types.

This implementation uses Binary Heap [28] Sorting algorithm for adding a new element to priority queue and removing an element from priority queue. In the priority queue elements are stored as structures which consist of nodeID, application ID, eligibility time and deadline and sorting is done based on deadline value of the queue element. There are two functions developed to access priority in the system.

5.1.1 enqueueGlobal()

First priority queue related function is enqueueGlobal() function. This function gets the new element and it stores it in priority queue based on its deadline value. It returns “0” after finishing enqueue process. If the queue is full then it returns “-1” to indicate caller application that the priority queue is full.

5.1.2 dequeueGlobal();

The second priority queue related function is dequeueGlobal() function. This function enters the priority queue and removes the head element of the priority queue

and returns it to caller thread. If the priority queue is empty then it returns NULL structure to notify caller thread.

5.2 Implementation of RTXX Protocol over Ethernet

Implementation of RTXX Protocol consists of several software modules that some of them are discussed in the Real Time Ethernet implementation in Chapter 3 .

In this part of the thesis, RTXX Protocol Implementation will be discussed in three sections;

- Real Time Ethernet Interface
- RTXX Protocol Core Implementation
- RTXX Protocol Application Interface

In *Section 5.2.1*, integration between Real Time Ethernet Implementation and RTXX Protocol Implementation will be discussed. In *Section 5.2.2*, RTXX Protocol Software Architechture and lifecycle behavior of implementation threads will be discussed. Finally in *Section 5.2.3*, interfaces that are provided for RTXX Protocol

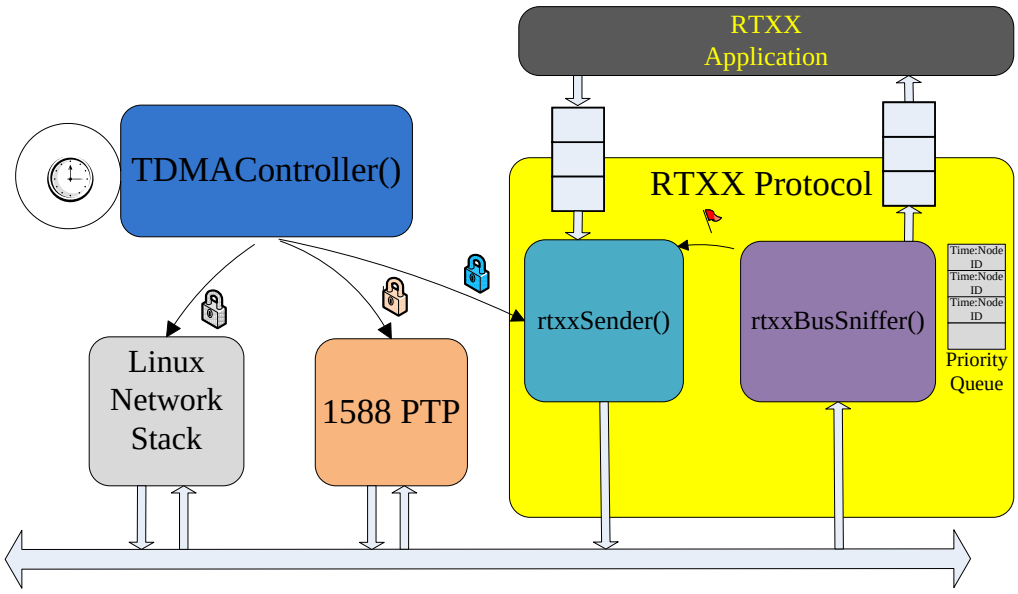

Figure 15: Software Architecture

Application Interface will be discussed.

Full software architecture including Real Time Ethernet Infrastructure is illustrated in *Figure 15*.

5.2.1 Real Time Ethernet Interface

RTXX Protocol needs to be controlled by TDMAController() thread in Real Time Ethernet implementation. TDMAController() thread have to create separate locking mechanisms for each custom protocol or application which needs to access Ethernet bus. For RTXX Protocol Implementation there is a new condition variable, named *condRTXX*, is created and added to TDMAController() function. Then a POSIX conditional variable locking mechanism for *condRTXX* variable implemented to the transmission thread of the RTXX Protocol Implementation. By this way TDMAController application can enable or disable transmission from RTXX protocol at anytime.

5.2.2 RTXX Protocol Core Implementation

RTXX Implementation contains 2 main threads to organize communication. These are rtxxBusSniffer() and rtxxSender(). Each of the function is described in the following sections.

5.2.2.1 rtxxBusSniffer() Thread

The main function of this thread is monitoring the incoming ethernet traffic and updating the system level the priority queue by the content of the captured packets. When an RTXX Protocol packet captured by this thread, its reservation in the priority queue is removed. Then rtxxBusSniffer() thread checks the destination address of the packet to understand if the packet belongs to its host or another host in the network. If the packet is for its own host, it sends the packet directly to the RTXX Protocol Application over POSIX message queues. Then it decomposes communication requests from the captured packet and updates the priority queue

with these new requests. After then it checks the head of the priority queue to analyze next packet sender's identification. If the sender identification belongs to computer which bus sniffer running on, it indicates the rtxxSender() thread by sending a signal. After that, rtxxBusSniffer()waits for a new packet to receive. Algorith for rtxxBusSniffer is illustrated in *Figure 16*.

5.2.2.2 rtxxSender() Thread

This thread is responsible to send RTXX Protocol packets to the Ethernet bus. It activity is controlled by the TDMAController() thread via POSIX mutexes and conditional variables. rtxxBusSniffer activates the rtxxSender() by sending a signal to it. After than rtxxSender() checks the time slot and if it is not real time slot it blocks itself with POSIX conditional variable. When the real time slot comes, TDMAController thread unblocks the TDMAController conditional variable and let rtXXSender() run. Then rtxxSender() thread gets the RTXX protocol packet from RTXX application via POSIX message queues, sends it to Ethernet bus and waits until the next signal receives. Algorithm for rtxxSender() thread is illustrated in *Figure 17*

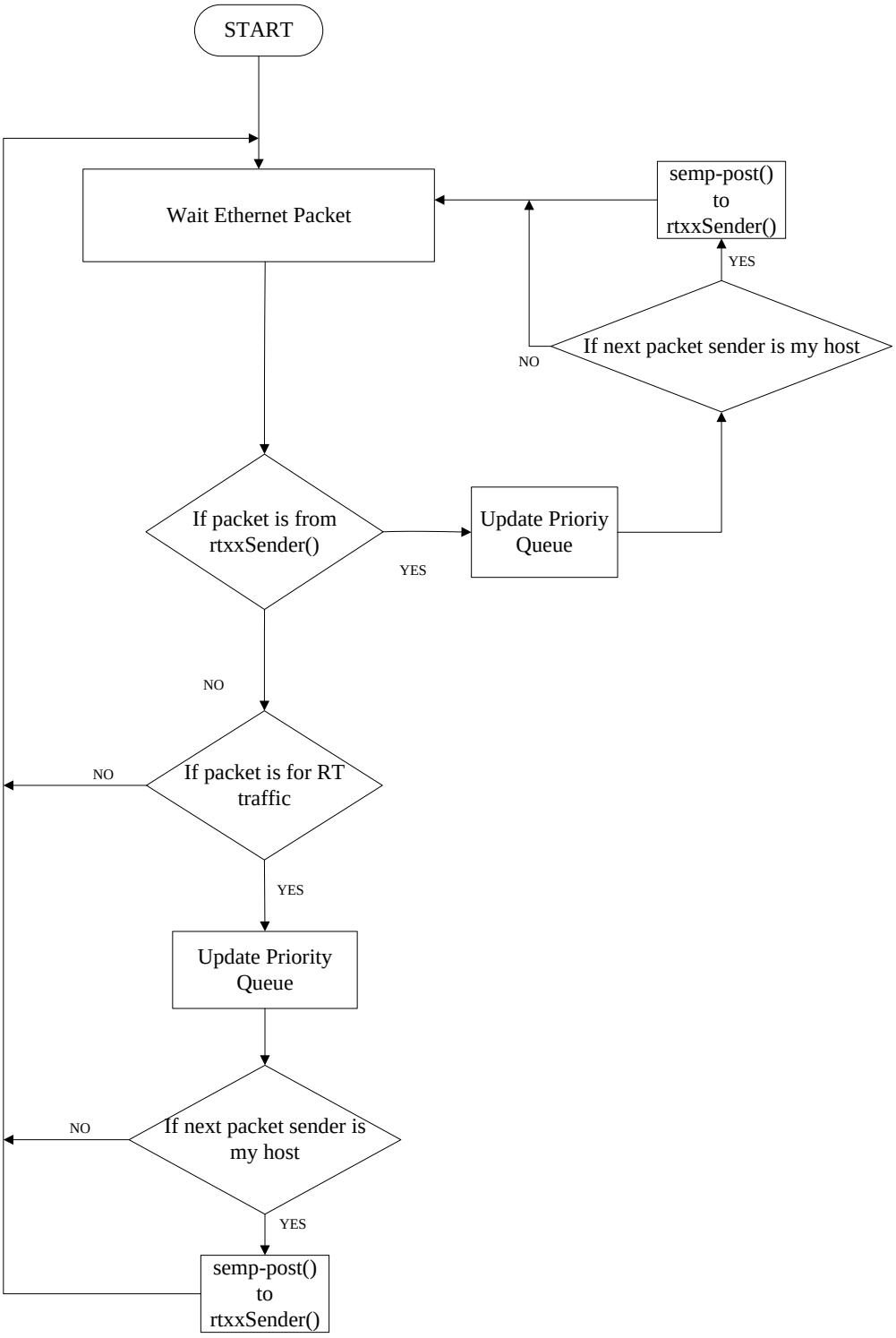

Figure 16: rtxxBusSniffer() Algorith

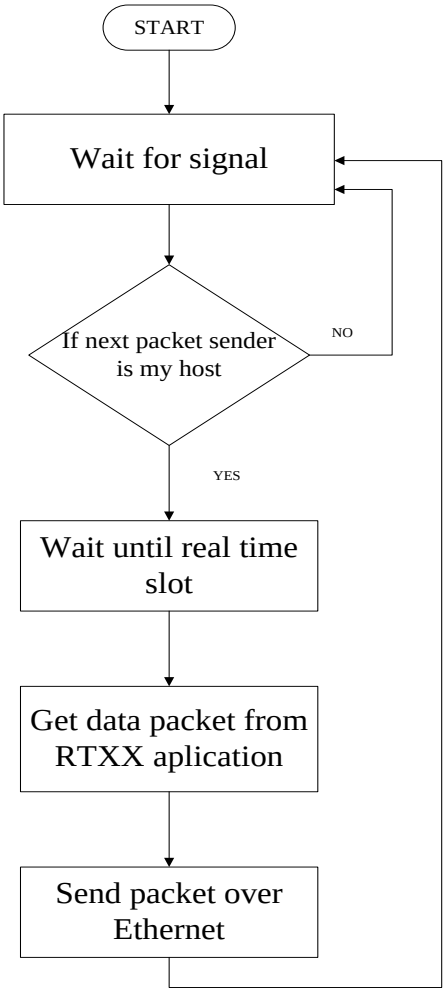

Figure 17: rtxxSender() Algorithm

5.2.3 RTXX Protocol Application Interface

RTXX Protocol Implementation provides user interfaces to RTXX Protocol applications. Protocol packet content creation and configuration is under responsibility of application. Packet buffering mechanism between applications and RTXX Protocol is also provided in this implementation via Priority Based Posix Message Queues.

This study only implements low level communication requirements for RTXX Protocol such as interfaces for accessing RTXX Protocol Layer, buffering and priority based packet transmission in RTXX Protocol.

5.2.3.1 recvfromRTXX()

Control applications can send and receive packets on RTXX Protocol via two application protocol interfaces (API). One of these interfaces is recvfromRTXX(). This API requires a buffer as an argument which the decomposed application data coming from the rtxxBusSniffer() thread will be stored in via application reception buffers which is the buffering mechanism between the receiver application and RTXX Protocol. Algoritm for recvfromRTXX() function is illustrated in *Figure 18*

5.2.3.2 sendtoRTXX()

The other API that RTXX Implementation offers is the sendtoRTXX() API. This function requires five arguments to run. These are destination host id, communication request buffer, communication request size, application data and application data size. Controller application can call sendtoRTXX() function after filling these arguments. After then, senttoRTXX() function generates a new Ethernet frame buffer based on these arguments and puts it into application transmission buffer queue which is the buffering mechanism between the application and RTXX Protocol. Afterward, rtxxSender() function takes the application data from transmission buffer queue and sends it through Ethernet Interface. Algorithm for sendtoRTXX() function is illustrated in *Figure 19*.

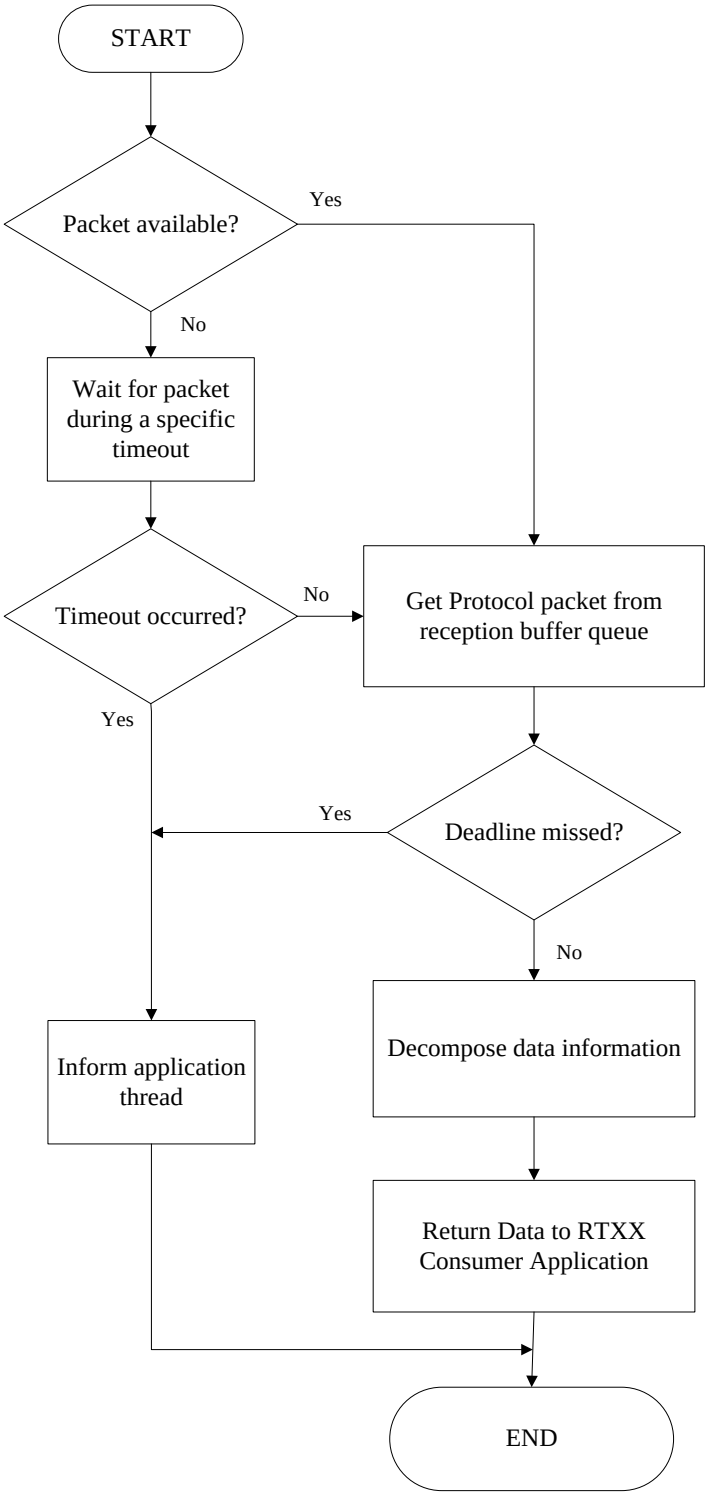

Figure 18:recvfromRTXX() Algorithm

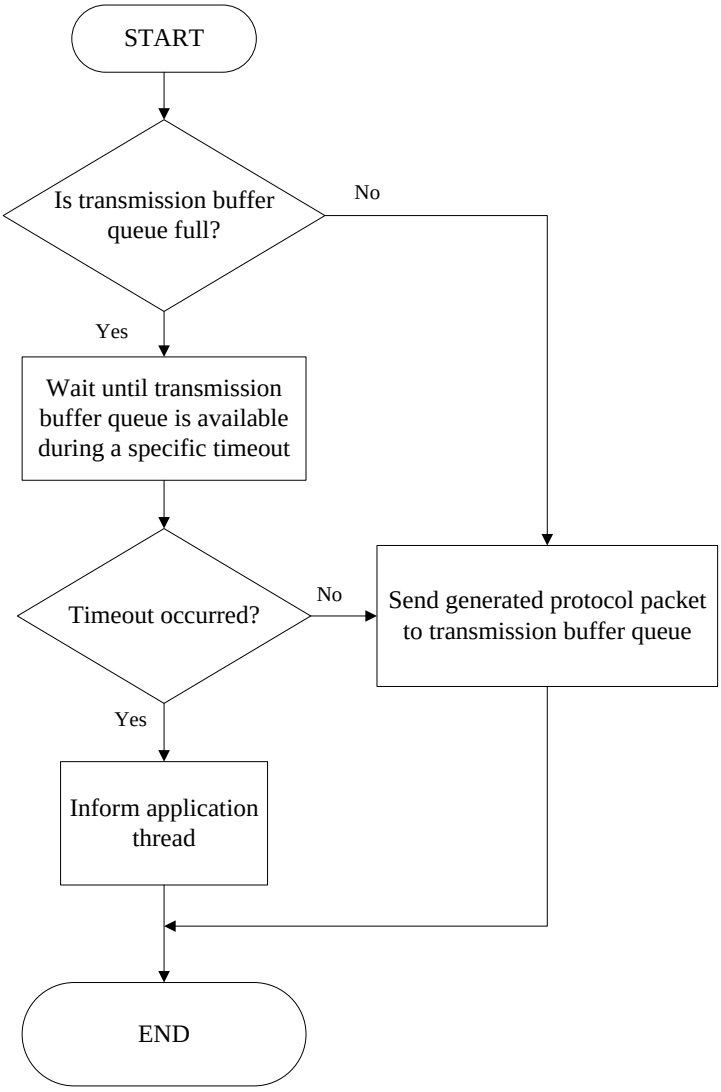

Figure 19: sendtoRTXX() Algorithm

CHAPTER 6

EXPERIMENTAL EVALUATION OF OUR IMPLEMENTATION

There are three protocol implementations made during the thesis study as IEEE 1588 PTP, Real Time Ethernet Protocol and RTXX Protocol. Although the implementation software passed from the software debugging process, for detecting possible delays on target and seeing the proper operation, additional target based experiments are required.

In the following sections there are several experiments for each protocol implementation. The experiments aim to measure the delays arising from each underlying mechanisms of protocols, detect and correct possible coding and designing faults. Because the protocols need to operate within time bounds, in other words real-time, determining the delays and their sources properly will help us to know the limits and applicability of the protocol implementations

Experiments are divided into 4 groups as Timing Experiments, Network Stack Experiments, IEEE 1588 PTP Experiments and RTXX Protocol Experiments

Timing Experiments which are Periodic Timer Accuracy Experiment, Slot Switching Latency Experiment and Periodic Timer Synchronization Accuracy Experiment measure latencies arising from the Operating System. These experiments show us the timing limits and capabilities of the Operating System Test results are collected for both Real Time and Non Real Time operations as scheduling, signaling and timing.

The Network Stack Experiment as Roundtirp Delay Experiment aims to measure the delays arising from Linux Network Stack. Each node in the Real Time Network use Linux network stack to manage incoming packet and outgoing packets. Therefore,

these experimental results help us determine the throughput of the protocol implementations.

IEEE 1588 PTP Experiment as IEEE 1588 PTP Time Synchronization Accuracy Experiment is done to measure the IEEE 1588 PTP synchronization accuracy in terms of packet size and scheduling type of the Operating System as real-time and non real-time. The results of this experiment determine the time drift between nodes in the Real Time Ethernet Network which affect the time slot synchronization accuracy and utilization of Real Time Network.

RTXX Protocol Experiments as RTXX Application Interface Latency Experiment, Queuing Latency Experiment, Real Time Traffic Experiments and Non Real-Time Traffic Experiment are done to verify the correct operation and measure latencies arising from RTXX Protocol implementation. These experimental results affect the deadline and eligibility parameter of RTXX Protocol Structure.

6.1 Experiment Environment

Experimental results were conducted with following hardware and software configuration:

- Intel® Atom™ Processor Z5xx Series and Intel® System Controller Hub US15W Development Kit
- Intel® 82574L Gigabit Ethernet Controller
- Open Suse 11.2 with/without Preempt-rt patch
- Cross over Ethernet cable

C programming language and POSIX programming standards are used to conduct test applications.

6.2 Experimental Results

6.2.1 Periodic Timer Accuracy

Periodic Timer Accuracy experiments were made in order to measure the accuracy and reliability of the timing mechanism of the real time network node. Experimental results are important to see the accuracy of the periodic timer which directly affects the true interoperability of the Real Time Ethernet and RTXX Protocol implementations. Rather than measuring the operability of the implementation, this experiment aims to measure the real time capability of the Operating System.

Periodic timer implementation determines the time slot interval accuracy. In this experiment, TDMAController() thread instance is generated for periodic timer creation to measure the time interval between sequential timer ticks. Measurements are collected respectively for 10.000Hz and scheduling types as real time and non real time. During the experiments CPU is loaded up to %99 with dummy functions to measure the worst case accuracy. For real time scheduling mode, the priority of the thread is determined as “99” which is the highest real time priority in Real Time Linux. The test is repeated 1000 times and graphs are generated based on these test results.

High resolution timers of Intel Architecture, as Time Stamp Counter and HPET, are used to measure the time interval between each sequential timer ticks. The following histograms show the distribution of time intervals between sequential clock ticks. *Figure 20* shows the measurements with non real time scheduling and *Figure 21* shows the measurements with real time scheduling.

When we look at the results, it is observed that there are delays at the microsecond level and real time threads have a deterministic and more consistent structure compared to the non real time ones which makes real time threads more suitable for TDMAController() implementation.

In Table 7 and Table 8 measurements are illustrated with the statistical information.

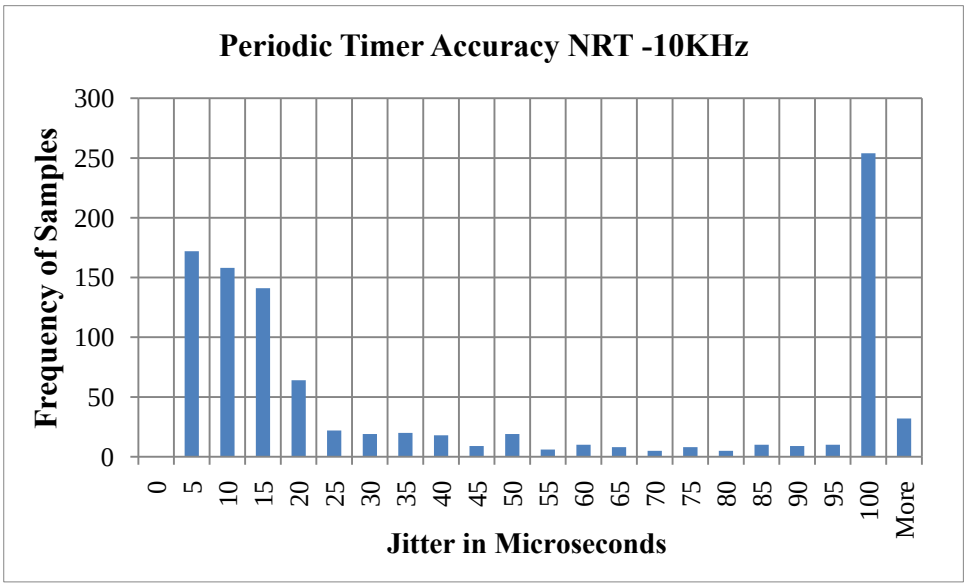

Figure 20: Periodic Timer Accuracy for 10KHz with Non Real Time Scheduling

Table 2: Experimental Results for Periodic Timer Accuracy for 10KHz with Non Real Time Scheduling

Number of Samples	Minimum	Maximum	Average	Standard Deviation	Confidence Int (%95)
1000	0	2976	48	111,111	6.89

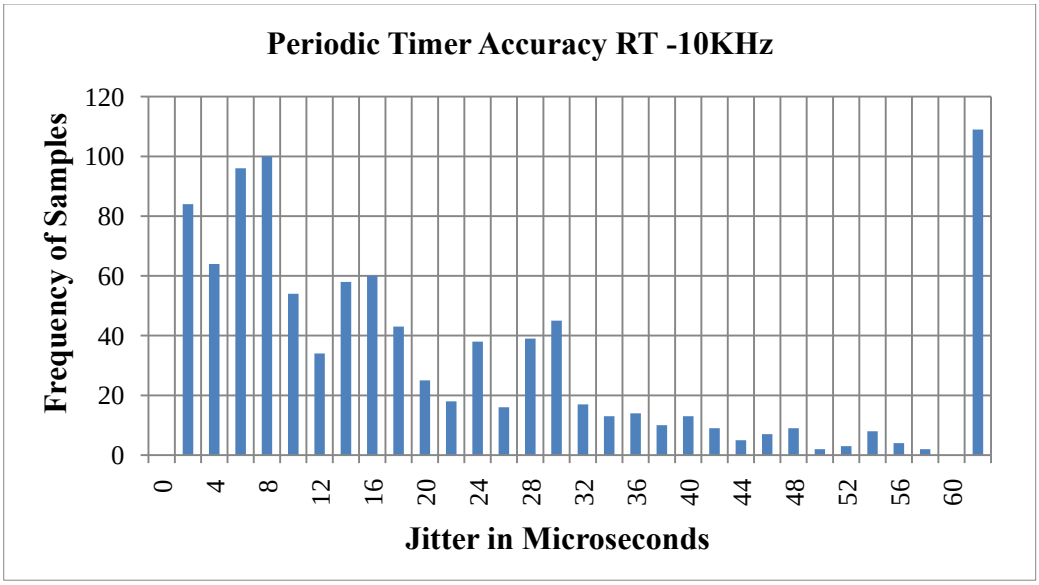

Figure 21: Periodic Timer Accuracy for 10KHz with Real Time Scheduling

Table 3: Experimental Results for Periodic Timer Accuracy for 10KHz with Real Time Scheduling

Number of Samples	Minimum	Maximum	Average	Standard Deviation	Confidence Int (%95)
1000	0	206	26	35,46	2,2

6.2.2 Slot Switching Latency:

Slot Switching Delay is the time interval between arrival of the timer signal and release of the specific software lock for related protocol. Slot Switching Delay experiments aim to measure the interval of the slot switching time which is the core of Real Time Ethernet implementation and the observing the changes for that switching time interval according to types of the scheduling such as real time and non real time. Experimental results will be effective in determining the time slot intervals in Real Time Ethernet implementation.

Measurements are taken with high resolution timers of Intel Architecture. *Figure 23* and *Figure 22* illustrate the slot switching delay for real time and non real time threads. In this experiment, TDMAController() thread is generated to activate periodic timer and signaling mechanism of RTXX Protocol. Test software output consists of a series of delay values. Each delay value is calculated from the differences between two time samples that the first sample captured at the reception of the timer signal and the second sample captured when the release of the software lock of determined protocol. Test software is executed 1000 times and the graphs are generated based on these test results

During the experiments CPU is loaded up to %99 with dummy functions. For real time scheduling mode, the priority of the thread is determined as “99”.

Experiments show that the slot switching latency values are almost the same for both real time and non real time scheduling. This stems from the size of switching code which is relatively small compared to other parts of the implementation. The experiment shows us that the real time scheduling performance does not have remarkable effects on slot switching delay.

We observe that the slot switching latency values are around 4500 Microseconds for both real time and non real time scheduling. and the jitter values varies a lot. This may arise from the interrupt latencies coming form peripherals as mouse, keyboard etc.Based on these delay values it's seen that slot switching delay has much lower effect on real time Ethernet implementation compared to other delay sources.

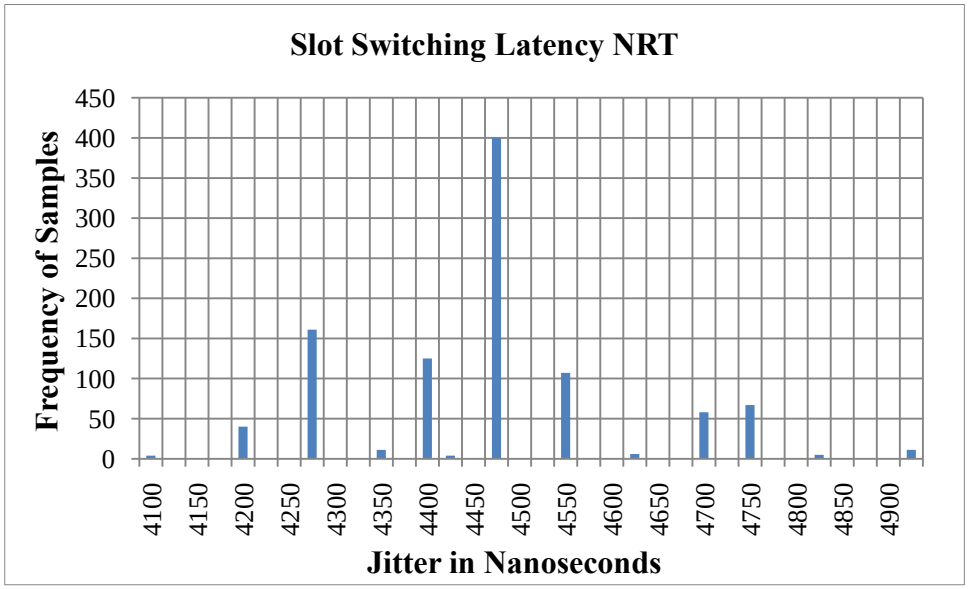

Figure 22: Slot Switching Latency with Non Real Time Scheduling

Table 4: Experimental Results for Slot Switching Latency with Non Real Time Scheduling

Number of Samples	Minimum	Maximum	Average
1000	3632	26260	4494

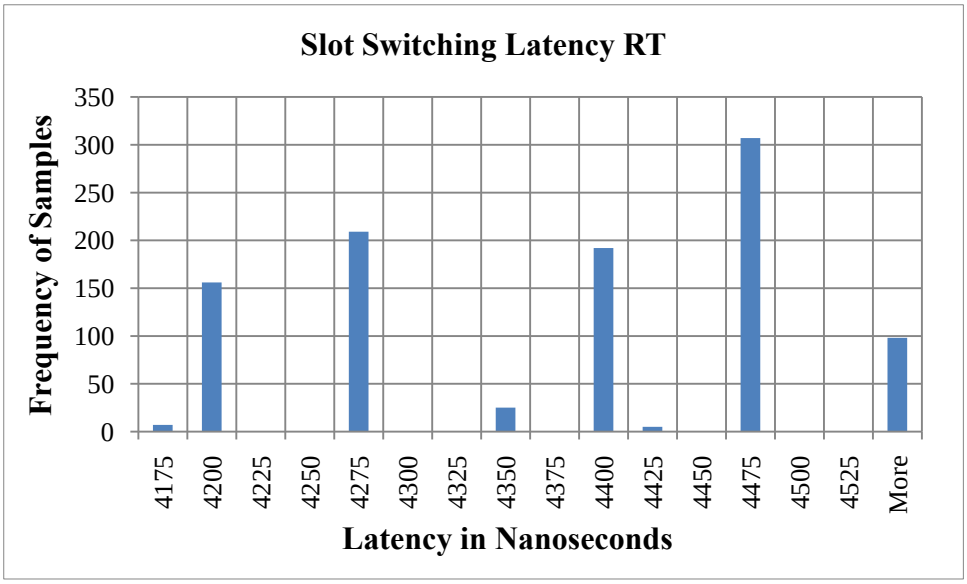

Figure 23:Slot Switching Latency with Real Time Scheduling

Table 5: Experimental Results for Slot Switching Latency with Real Time Scheduling

Number of Samples	Minimum	Maximum	Average
1000	4050	22978	4473

6.2.3 Periodic Timer Synchronization Accuracy

Periodic timer synchronization accuracy is another mechanism that influences the slot switching time in the implementation. Periodic Timer Synchronization Algorithm aims to synchronize periodic timer startup time of each RTXX Node after the synchronization of the system time with IEEE 1588 PTP Implementation. The difference between periodic timer accuracy experiments and periodic timer synchronization accuracy experiments is, periodic timer accuracy experiments aim to measure the periodicity of the timer, on the other hand, periodic timer synchronization accuracy experiments aims to measure the time difference between periodic timer startup times in master and slave node.

In this experiment, the aim is to measure the accuracy of the periodic timer synchronization algorithm. In the current implementation, besides synchronizing the system time, synchronized timer signal generation is required for simultaneous slot switching in the real time network. In the implementations found in the literature for

synchronized clock generation, the clock inputs are being used to create a shared clock and this necessitates special hardware elements. However in this implementation special hardware should not be required and standard Network Interface Cards should be used. Because of this, unlike the other implementations in the literature, software solutions are considered to assure the periodic timer synchronization. The first method that comes to mind is using the Ethernet packets as a trigger and starting the periodic timers after receiving the packets. But this method is not applicable when the aimed few microsecond sensitivity is considered because of the delays resulting from the transmission of the packet on a physical environment and Master and Slave nodes' network stacks.

In the system developed, from the start of the 2nd second of system time after making the system clock synchronization with IEEE 1588 PTP, trials are performed at 100 microsecond periods until the value of deviation is obtained that has a lower value then the predetermined deviation value. The number of the trials are increased or decreased according to the determined deviation value. For example, one trial may be enough for a 10 microseconds deviation value whereas tens or hundreds of trials may be required for a 1 microsecond deviation depending also on the processor's speed.In this experiment, repeating period is determined as 100 microseconds.

TDMAController() thread with Periodic Timer Synchronization Algorithm is spawned to measure periodic timer synchronization delays in the system. Test software outputs consist of a series of deviation values. Each deviation value illustrates the time deviation from predetermined synchronization point. Test software is executed 1000 times and the graphs are generated based on these test results.

Figure 24 shows the distribution of deviations from predefined synchronization time point for non real time scheduling. There are some peak points on the right side of the graphs which makes it unsuitable for this implementation. If we compare minimum and maximum values of the measured accuracy, we can see very big difference arising from the non real time scheduling. *Figure 25* shows the

distribution of delay deviations from predefined synchronization time point for real time scheduling. That figure has a uniform distribution of delay deviation around 1-2 Microseconds. Based on test results, periodic timer synchronization delays can be said to be sufficient for Real Time Ethernet implementation.

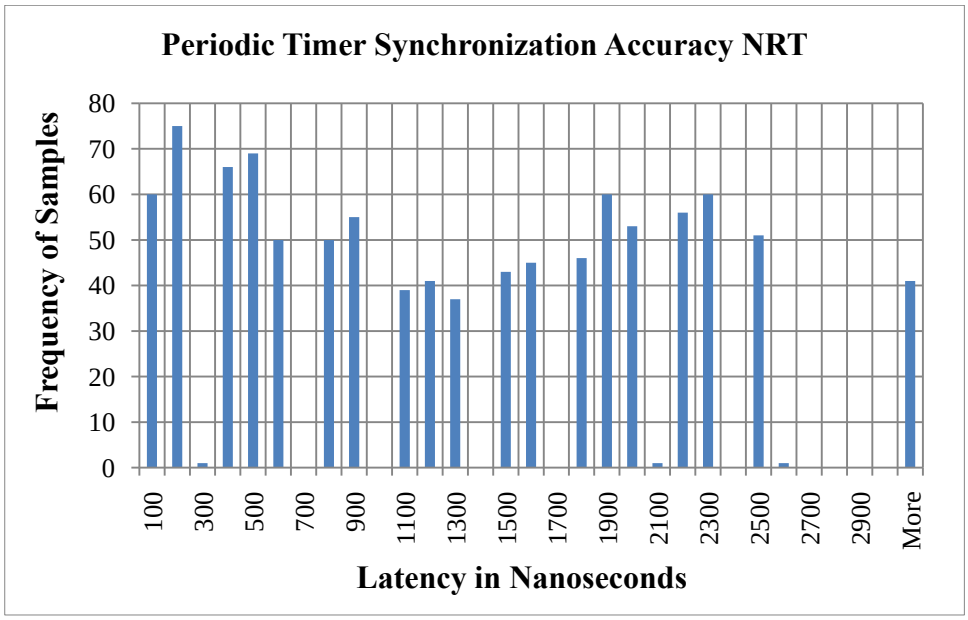

Figure 24: Periodic Timer Synchronization Accuracy for with Non Real Time Scheduling

Table 6: Experimental Results for Periodic Timer Synchronization Accuracy with Non Real Time Scheduling

Number of Samples	Minimum	Maximum
1000	35	25111503

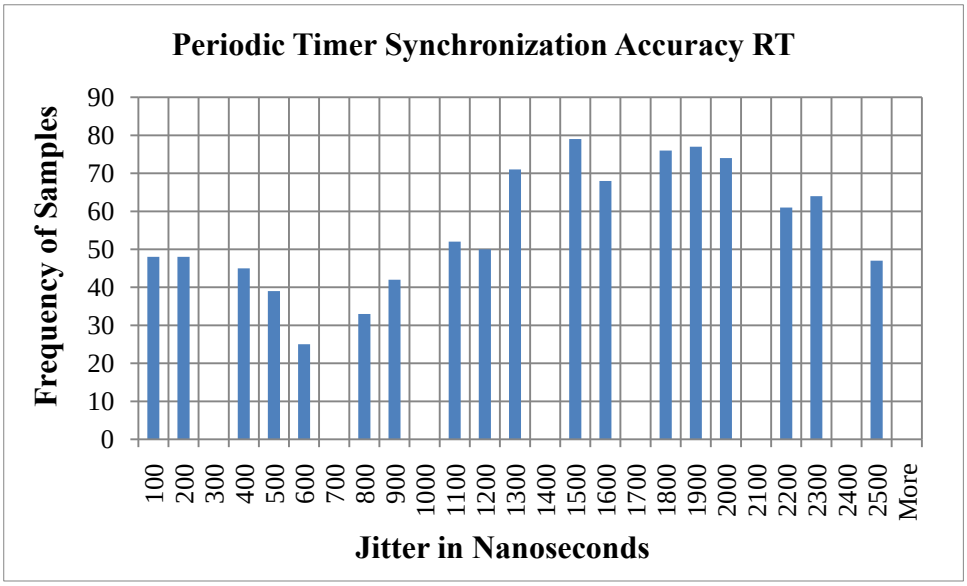

Figure 25: Periodic Timer Synchronization Accuracy for with Real Time Scheduling

Table 7: Experimental Results for Periodic Timer Synchronization Accuracy with Real Time Scheduling

Number of Samples	Minimum	Maximum	Average	Standard Deviation	Confidence Int (%95)
1000	35	2409	1349	694	43,04

6.2.4 Roundtrip Latency

The purpose of this experiment is to determine the delays that are resulting from the network stack of the operating system. For this purpose, a test software is developed in which a connection is set between two nodes with cross cable and the roundtrip latencies of Ethernet packets are recorded.

When developing the real time Ethernet and RTXX protocol implementations it is assumed that the whole communication network is shared because of this the implementation software is developed accordingly. When we look at the shared medium communications in general, we see that the receive and transmit channels are both connected to the same transmission channel and so every packet that is sent to the shared medium is also detected by the receiver channel.

The RTXX implementation owing to the same shared medium communication principle needs every outgoing packet's header data to update the priority queue in the protocol implementation. Because converting the standard Ethernet connection to shared ethernet requires hardware changes like shorting transmission lines with reception lines, considering the line emphedance and signaling issues, RAW socket background is used for packets that are sent to the system in the test software. If there is a raw socket that is listening on the same computer, a copy of every packet that is delivered outside is also sent to other raw sockets. So, like the shared medium communications, every packet that is sent is also recognized by the receiver channel and the actions can be taken accordingly.

When determining the time slot interval, the delays resulting from the network stack should also be taken into account because of the RAW socket usage in the Real Time Ethernet and RTXX protocol implementations. Round trip delay experiments are measured according to both the packet size and the scheduling type and the graphical representation of the experiment results are prepared.

During the experiment, a dummy thread is developed which will keep the system completely busy to reflect the potential changes on the system load and experiments are started. The test software has been run a thousand times and every delay value is recorded and represented on the graphics.

When we look at the test results, we observe that the packet size and the delay change as directly proportional to each other as expected. When the scheduling type is considered, we can conclude that the delay times are variable and inconsistent for non real time threads but they show a smooth distribution for the real time threads. When we consider the real time 60 byte packets, average roundtrip delay value is seen to be 729 microseconds. When 1 Gbit Ethernet connection is used, the theorical transmission time is about 0,5 microseconds in the physical medium. In this case, the delay value resulting from the network stack is find as:

Roundtrip Delay = 2 x (Transmit Delay + Propagation Delay + Reception Delay)

729 = 2 x (Transmit Delay + 0 + Reception Delay)

364,5 = Transmit Delay + Reception Delay

In this experiment Propagation Delay is not included in the calculation above because the delay value of propagation on the transmission line is very low, between 3.71 to 8.34 Nanosecond/Meter [29], compared to other delay sources in the implementation. It is considered that propagation delay has not any remarkable effect on performance of the Real Time Ethernet and RTXX Protocol implementation.

It can be concluded that the required time for a single packet to be transmitted and received in the real time Ethernet and RTXX protocol implementations should be minimum 365 Microseconds for above test conditions. When determining the time slot interval this value should be considered.

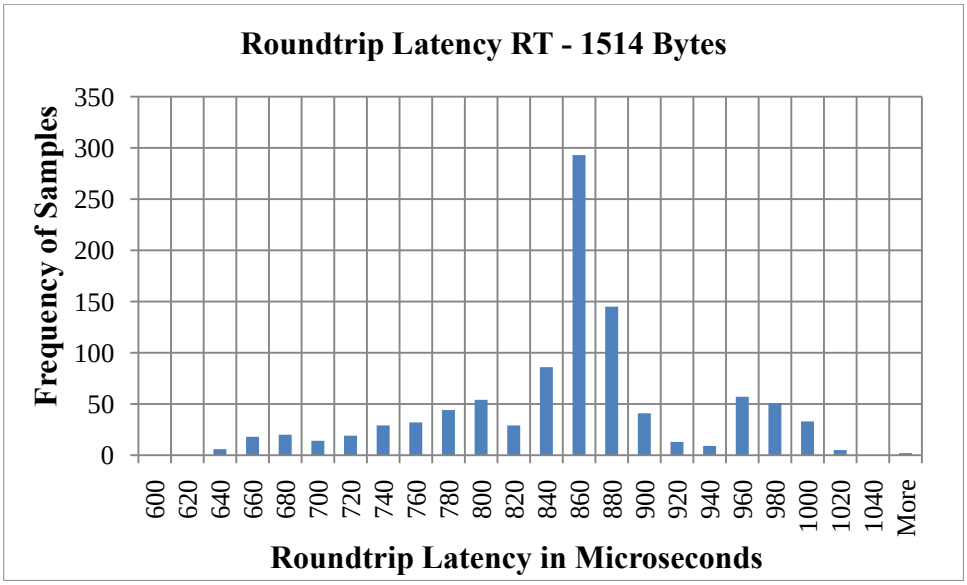

Figure 26: Roundtrip Latency for 1514 Byte Packets with Real Time Scheduling,

Table 8: Experimental Results for Roundtrip Latency for 1514 Byte Packets with Real Time Scheduling

Number of Samples	Minimum	Maximum	Average	Standard Deviation	Confidence Int (%95)
1000	639	1152	844	78,18	4,84

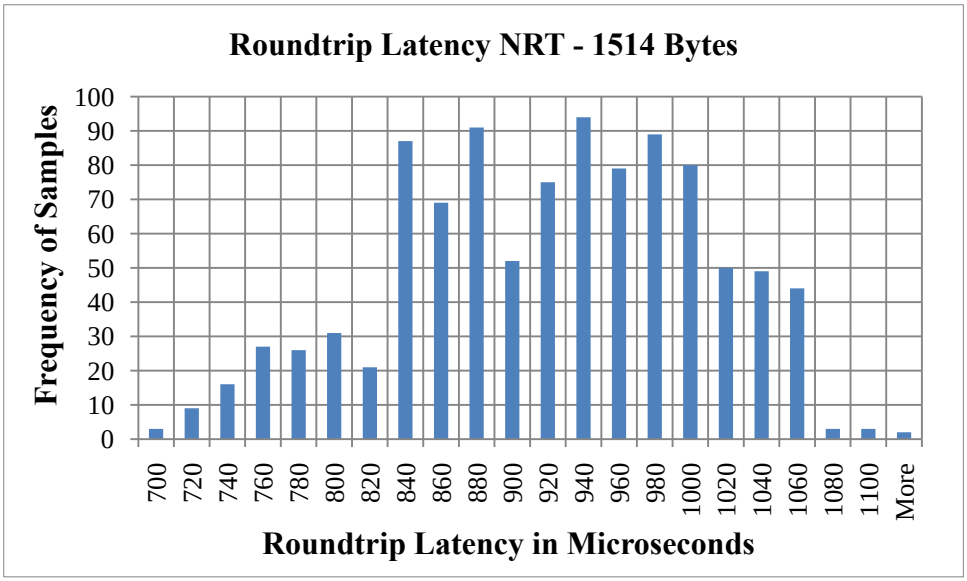

Figure 27: Roundtrip Latency for 1514 Byte Packets with Non Real Time Scheduling

Table 9: Experimental Results for Roundtrip Latency for 1514 Byte Packets with Non Real Time Scheduling

Number of Samples	Minimum	Maximum	Average	Standard Deviation	Confidence Int (%95)
1000	665	1103	911	84,9	5,26

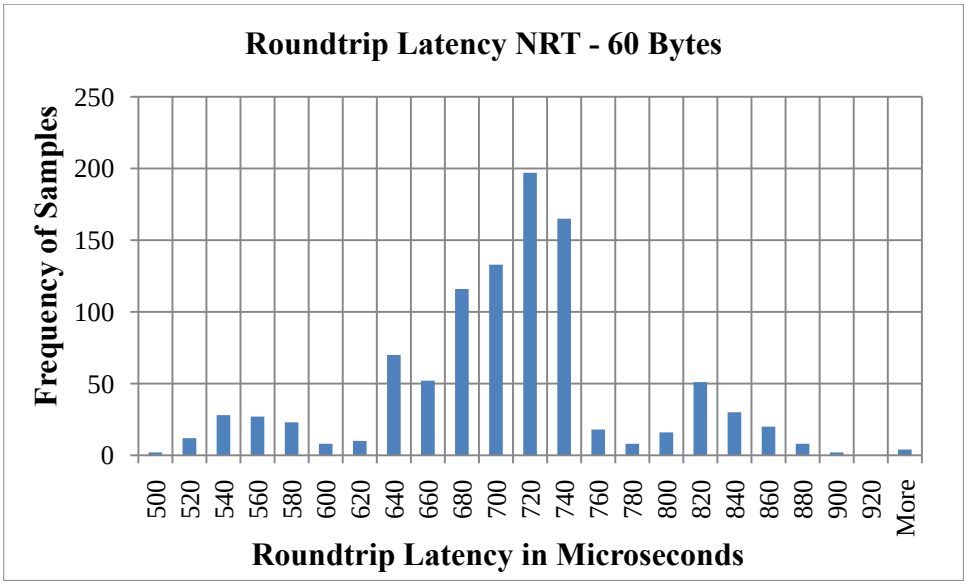

Figure 288: Roundtrip Latency for 60 Byte Packets with Non Real Time Scheduling

Table 10: Experimental Results for Roundtrip Latency for 60 Byte Packets with Non Real Time Scheduling

Number of Samples	Minimum	Maximum	Average	Standard Deviation	Confidence Int (%95)
1000	495	1551	697	79,67	4,94

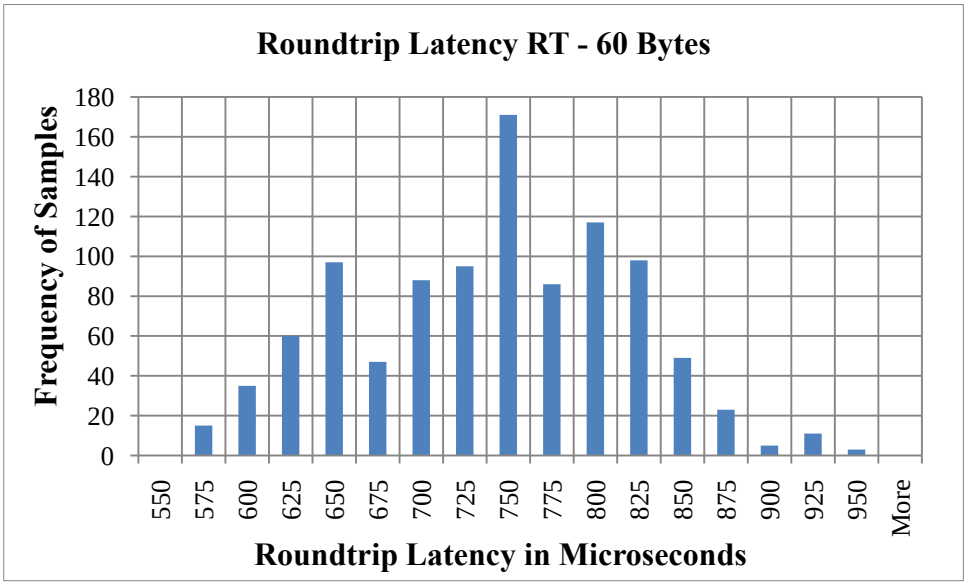

Figure 29: Roundtrip Latency for 60 Byte Packets with Real Time Scheduling

Table 11: Experimental Results for Roundtrip Latency for 60 Byte Packets with Real Time Scheduling

Number of Samples	Minimum	Maximum	Average	Standard Deviation	Confidence Int (%95)
1000	557	931	729	77,39	4,49

6.2.5 IEEE 1588 Time Synchronization Accuracy

Time synchronization over IEEE 1588 is the main requirement for both Real Time Ethernet and RTXX Protocol Implementations. Success of the implementations depends on synchronization accuracy of IEEE 1588 PTP Implementation. This experiment aims to measure the time drifts between Master Node and Slave Node in Real Time Network after the IEEE 1588 PTP synchronization algorithm is executed.

In this experiment, TDMAController() thread was generated on both Master and Slave Node and IEEE 1588 PTP protocol algorithm was executed for time synchronization. This process was repeated 1000 times sequentially and time

differences between Master and Slave node, called offset, were recorded. Test software output consists of these offset values.

Test software is executed for different scheduling types as real time and non-real time and different packet sizes as 60 Byte and 1514 Bytes and offset values for these experiments are illustrated in graphs. *Figure 31* and *Figure 30* illustrates the synchronization offset distribution for 60 Byte Ethernet packet size for real time and non-real time scheduling. *Figure 31* and *Figure 32* illustrates the synchronization offset distribution for 1514 Byte Ethernet packet size for real time and non-real time scheduling. Figures show that packet size does not have remarkable effect on the synchronization accuracy. On the other hand, scheduling has effect on synchronization accuracy that real time scheduling has much more deterministic results compared to non real time scheduling

With this test conditions average IEEE 1588 PTP synchronization drift is about 35 Microseconds and worst-case synchronization drift seems 166 Microseconds for real time scheduling. For non real time scheduling, difference between the maximum and minimum values of the synchronization accuracy is too big so non real time scheduling does not acceptable for this implementation. During the time slot calculation this drift must be considered and time slots should have big enough to tolerate this synchronization jitter.

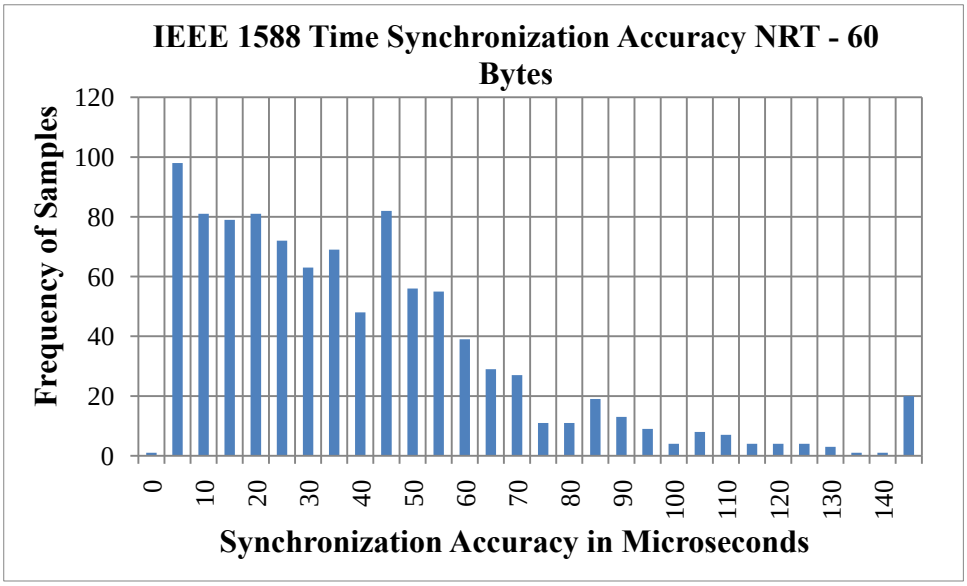

Figure 30: IEEE 1588 Time Synchronization Accuracy for 60 Byte Packets with Non Real Time Scheduling

Table 12: Experimental Results for IEEE 1588 Time Synchronization Accuracy for 60 Byte Packets with Non Real Time Scheduling

Number of Samples	Minimum	Maximum	Average
1000	0	500308	1043

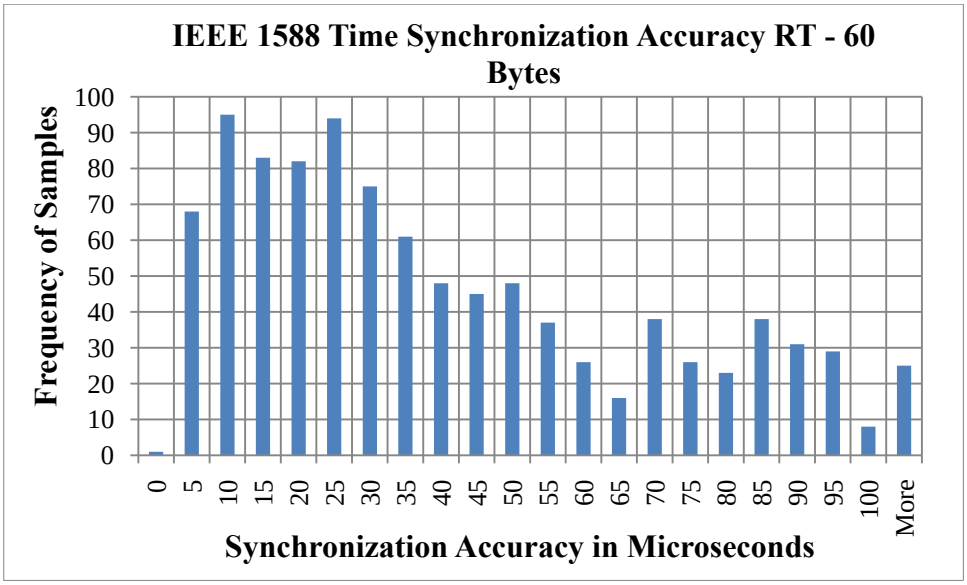

Figure 31: IEEE 1588 Time Synchronization Accuracy for 60 Byte Packets with Real Time Scheduling

Table 13: Experimental Results for IEEE 1588 Time Synchronization Accuracy for 60 Byte Packets with Real Time Scheduling

Number of Samples	Minimum	Maximum	Average	Standard Deviation	Confidence Int (%95)
1000	0	166	38	28,72	1,78

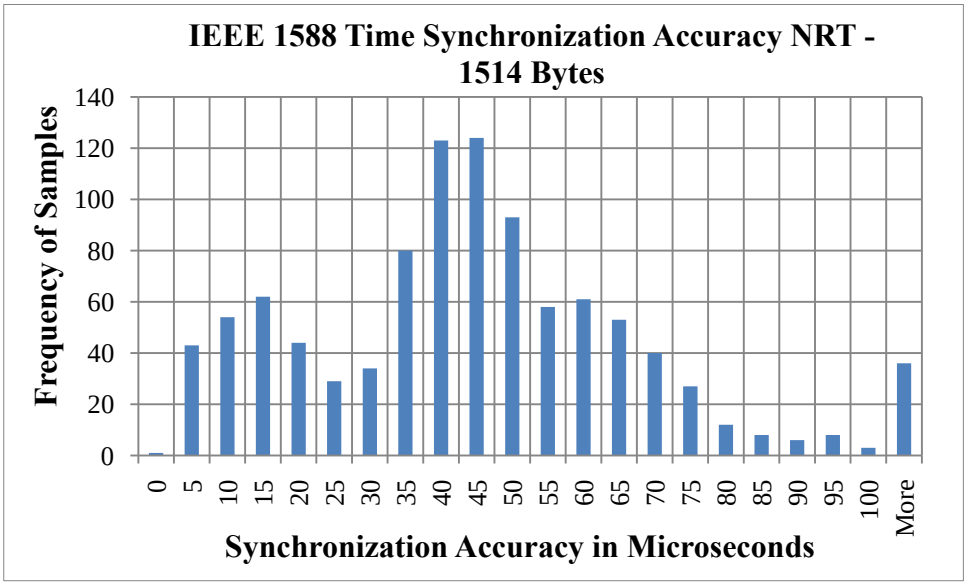

Figure 32: IEEE 1588 Time Synchronization Accuracy for 1514 Byte Packets with Non Real Time Scheduling

Table 14: Experimental Results for IEEE 1588 Time Synchronization Accuracy for 1514 Byte Packets with Non Real Time Scheduling

Number of Samples	Minimum	Maximum	Average
1000	0	500029	1024

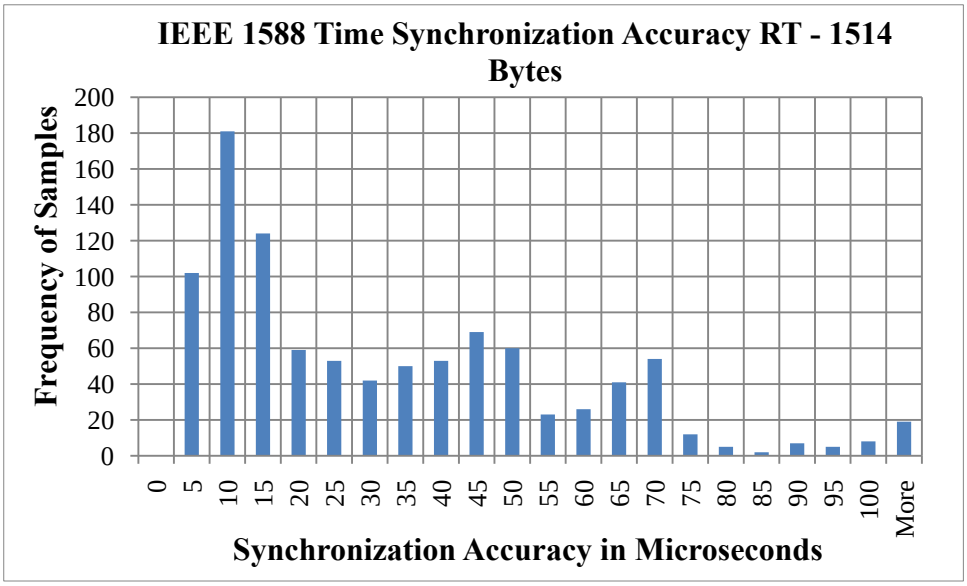

Figure 33: IEEE 1588 Time Synchronization Accuracy for 1514 Byte Packets with Real Time Scheduling

Table 15: Experimental Results for IEEE 1588 Time Synchronization Accuracy for 1514 Byte Packets with Real Time Scheduling

Number of Samples	Minimum	Maximum	Average	Standard Deviation	Confidence Int (%95)
1000	0	164	31	27,01	1,677

6.2.6 RTXX Application Interface Latency

Another latency source in the protocol implementation is RTXX Application Interface Latency. This latency is arising from the aplication interfaces that an applicaiton uses for accessing RTXX Protocol engine. As mentioned in *Section 5.2.3*, there are some buffering mechnisms at application interfaces which uses Posix Message Queues. The following experiments aims to measure the overall latency results from buffering and computation at user interface layer.

During the experiments two types of latecies were mesured in the system. First one was the application to RTXX Protocol latency, and the second one was RTXX Protocol to application latency. Message queue size was determined as 60 Bytes in

the experiments that the simulations shows that this size is sufficient for RTXX Protocol messaging [30].

In the experimental results, CPU is loaded up to %99 with dummy functions to measure the worst case latency. For real time scheduling mode, the priority of the thread is determined as "99". Test is repeated 1000 times and *Figure 34* and *Figure 35* are generated based on these test results.

It is seen that the latency values resulting from RTXX Protocol to RTXX application and RTXX application to RTXX Protocol is almost the same. On the other hand, latency values varies for real-time and non real-time scheduling which the values are illustrated in *Table 35* and *Table 34*. The results show that the real time scheduling gives more deterministic and lower latency values compared to non real-time scheduling. The worst-case latency is determined as 90 Microseconds that should be considered during the slot interval determination of RTXX Protocol

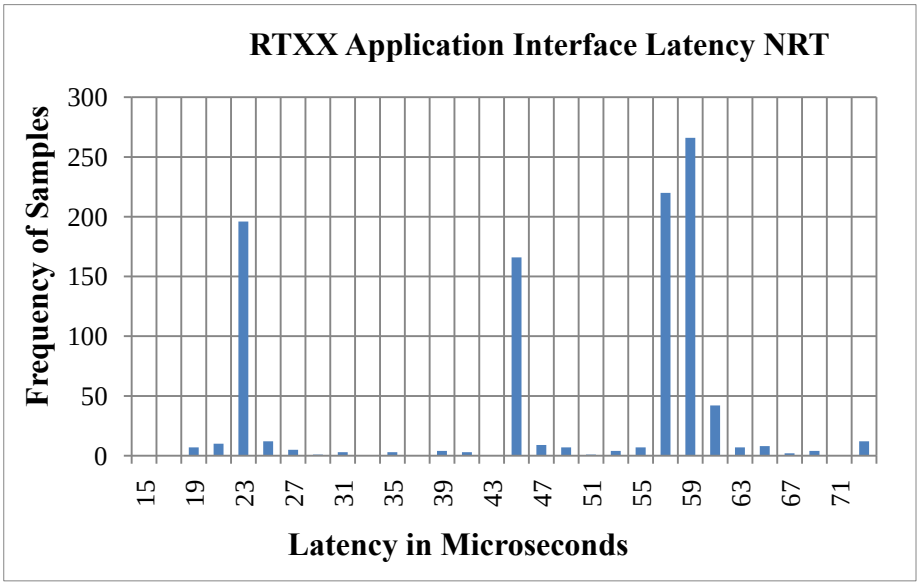

Figure 34: RTXX Application Interface Latency with Non Real Time Scheduling

Table 16: Experimental Results for RTXX Application Interface Latency with Non Real Time Scheduling

Number of Samples	Minimum	Maximum	Average	Standard Deviation	Confidence Int (%95)
1000	17	313	47	17,51	1,09

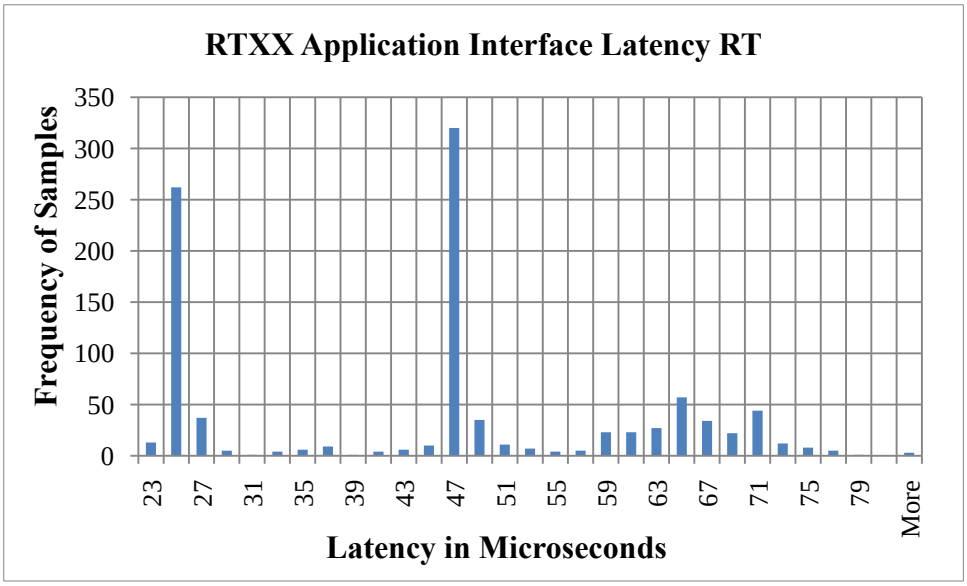

Figure 35: RTXX Application Interface Latency with Non Real Time Scheduling

Table 17: Experimental Results for RTXX Application Interface Latency with Non Real Time Scheduling

Number of Samples	Minimum	Maximum	Average	Standard Deviation	Confidence Int (%95)
1000	11	90	44	16,195	1

6.2.7 Queuing Latency

Priority queue implementation is the main requirement for the RTXX Protocol. Because of this priority queue update delay is another factor that may affect the width of Real-time slot interval. Because the update process of the priority queue is repeated for each incoming and outgoing packet realted to RTXX protocol, knowing the update delays is important to determine the RTXX real time slot interval in the network. Few microseconds update latency is expected for this implementation.

There are two functions, named enqueueGlobal() and dequeueGlobal, in the software which interracts with priority function directly. enqueueGlobal() function is responsible to add new element into the priority queue and dequeueGlobal() is responsible to remove the smallest deadlined element in the priority queue. Because

the Binary Heap algorithm used in priority queue adaptation update time of the queue may varies according to the value of each added and removed element and number of elements in the queue.

In the experiments maximum number of elements in the Priority Queue size is determined as 1000 for the test software. Firstly, the queue is filled with new elements by calling enqueueGlobal() 1000 times. After that the dequeueGlobal() functions called 1000 times to clear the the queue completely and each of enqueueGlobal() and dequeueGlobal() function time consumptions are recorded. To measure the worst-case latency of the priority queue access with enqueue and dequeue functions, each new element's value that will added to the queue is dicreased. During the experiments CPU is loaded up to %99 with a dummy functions to measure the worst case access delay to the queue. Experiment were repeated for both real time and non real time scheduling. For real time scheduling mode, the priority of the thread is determined as "99".

Previous studies for the RTXX Protocol shows that the maximum required element number for the priority queue is about 32. Experimental worst-case queuing delay for 32 elements for Real Time scheduling is about 4 Microseconds in this implementation. Although the queuing latencies have not much influence in overall latency, it should be considered during the determination of time slot interval of RTXX Protocol.

6.2.8 Real Time Traffic Experiments

In experiments so far, infrastructural measurements have been taken for Real Time Ethernet and RTXX Protocols. In this and subsequent experiments, measurements will be taken at system level. Previously collected measurements will be used to obtain time slot size and frequency of RTXX protocol.

For the determination of the delay parameters and their values in the system, real time scheduling will be used to generate protocol threads. Thus the experimental results which are collected with the real time scheduling will be used.

Table 18 illustrates the summary of the real time experimental results.

Table 18:Summary of Real Time Scheduling Experimental Results

Delay Parameter	Minimum	Average	Maximum
Periodic timer accuracy	0	26	206
Slot switching latency	4	4,5	23
Periodic timer synchronization accuracy	0	1,5	2,5
Network Stack Transmission Delay	279	365	466
IEEE 1588 Time Synchronization accuracy	0	38	166
RTXX Application interface latency	11	44	90
Queuing Latency	4	4	4

Considering these values, it can be seen that the best case latency for overall system is 298 Microseconds, Average latency is 483 Microseconds and the worst-case latency is 957 microseconds. Some delays may occur more than one node, because of this determining the value of the slot interval as the range of 1 Millisecond will not be wrong. In this case maximum frequency for the real time slot will be 1000 Hz. During the experiments Two Intel® Atom™ Processor Z5xx Series and Intel® System Controller Hub US15W Development Kit connected each other with a cross cable and Developed Real Time Ethernet and RTXX Protocol application test software installed both PC. First computer named as N_1 and the second computer named as N_2. The following statements explains the operation of test software.

- The initialization process is started by 1588 PTP Time Synchronization process.

- After the time synchronization N_1 sends an RTXX job including (N_2,*d*,*0*,*T*) request.
- N_2 captures the job which is sent by the N_1 and puts to request to its priority queue.
- Because the eligibility time is 0, N_2 immediately sends its first message that includes (N_2,*d*,*0*,*T*) request and the timestamp of the system time . By sending (N_2,*d*,*0*,*T*) request, N_2 reserves shared medium for itself for future messages.
- N_1 captures the packet coming from N_2, calculates the time difference between its system time and the time instance located in the packet and records it.
- N_2 continues to send (N_2,*d*,*0*,*T*) request with the message periodically during the lifecycle of test software.
- After a specific number of samples collected test software finalizes itself.

For the first RTXX Protocol experiment, time slot interval is determined as 1 Millisecond which is the smallest applicable time slot for this platform, deadline, *d*, is determined as 1 Millisecond, eligibility time, *e*, is determined as 0, Qmax is determined as 1 and the maximum message size in the system, F_{max}, is determined as the minimum Ethernet frame length 60 Bytes = 480 bits. In this conditions the theoretical minimum tolerable frequency value is as follows;

$$r_{min} = (Q_{max}+1) / (d_{min}-e_{max}) \qquad (1)$$

$$r_{min} = (1+1) /(0{,}001-0)=2000 \text{ Hz} \qquad (1)$$

And the theoretical bandwidth for this experiment is computed as follows;

$$r_{min} \text{ x } F_{max} = 2000 \text{ x } 480 = 960000 \text{ bit per seconds}$$

The overall bandwidth for the system is 1Gbps so the theoretical bandwidth requirement is less than one percent of the overall bandwidth. On the other hand, applicable time slot frequency for real time implementation can be at most 1000Hz and even in this frequency, the bandwidth usage is hundred percent of overall bandwidth because of the software delays. The transmission schedule for the first experiment is illustrated in *Figure 36.* The first set of experimental results illustrated in *Figure 37* that shows the latency distribution of packets sent by N2 in Microseconds level.

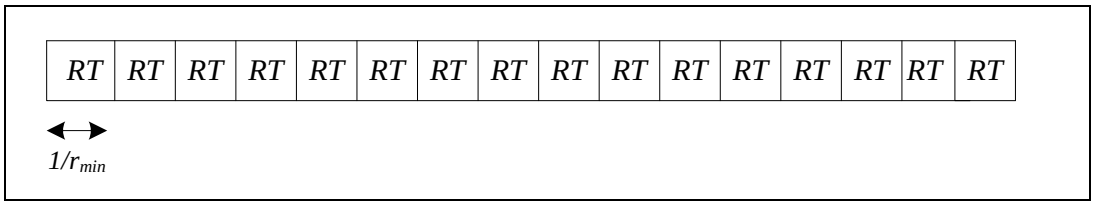

Figure 36: Transmission Schedule for Real Time Traffic Experiment 1

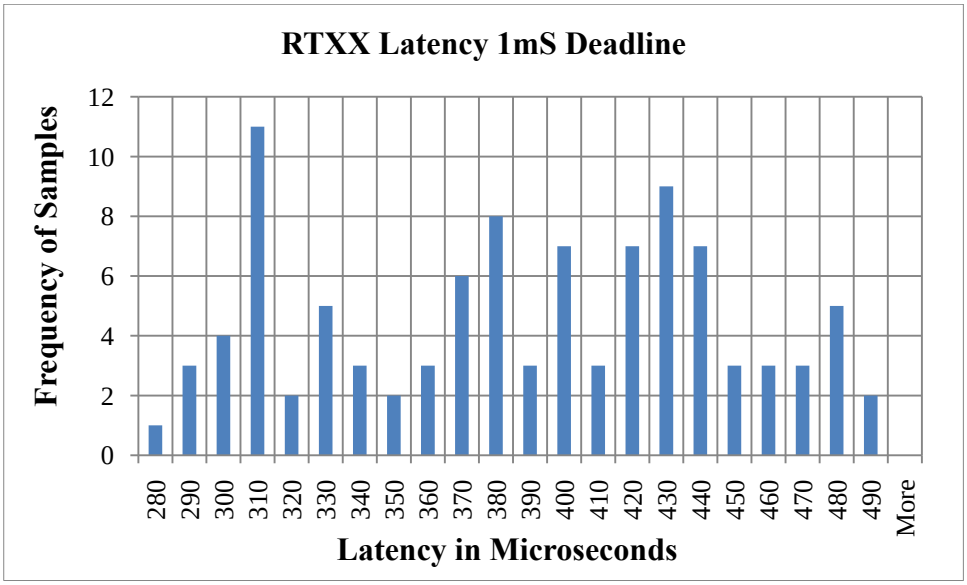

Figure 37: RTXX Latency for 1mS Deadline for Real Time Scheduling

Table 19: Experimental Results RTXX Latency for 1mS Deadline for Real Time Scheduling

Number of Samples	Minimum	Maximum	Average	Standard Deviation	Confidence Int (%95)
100	280	483	382	57,98	11,36

It can be seen that, for 1 Millisecond deadline, all of the experimental results are less than 1 Millisecond which means that the deadline requirement was met and the protocol worked fine during the experiment.

In the second RTXX Protocol experiment, time slot interval is determined as 1 Millisecond which is the smallest applicable time slot for this platform, deadline, *d*, is determined as 4 Millisecond, eligibility time, *e*, is determined as 0, Qmax is determined as 1 and the maximum message size in the system, F_{max}, is determined as the minimum Ethernet frame length 60 Bytes = 480 bits. In this conditions the theoretical minimum tolerable frequency value is as follows;

$$r_{min} = (Q_{max}+1) / (d_{min}-e_{max}) \qquad (1)$$

$$r_{min} = (1+1) /(0{,}004-0)=500 \text{ Hz} \qquad (1)$$

And the theoretical bandwidth for the second experiment is computed as follows;

$$r_{min} \text{ x } F_{max} = 500 \text{ x } 480 = 240000 \text{ bits per seconds}$$

The highest applicable frequency for the experiment is 250Hz for this platform because of the time slot interval of 1 Millisecond. The transmission schedule for the second experiment is illustrated in *Figure 38* The second set of experimental results illustrated in *Figure 39* that shows the latency distribution of packets sent by N_2 in Microseconds level.

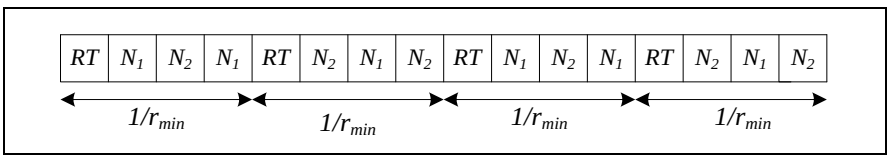

Figure 38:Transmission Schedule for Real Time Traffic Experiment 2

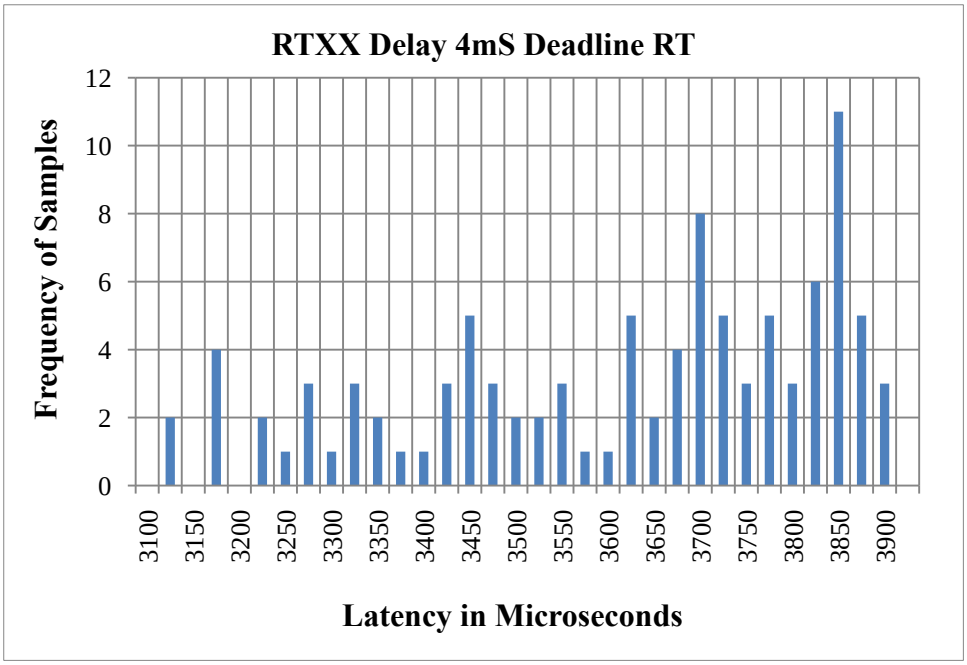

Figure 39: RTXX Latency for 4mS Deadline for Real Time Scheduling

Table 20: Experimental Results RTXX Latency for 4mS Deadline for Real Time Scheduling

Number of Samples	Minimum	Maximum	Average
100	3112	3879	3600

It can be seen that, for 4 Millisecond deadline, all of the experimental results are less than 4 Millisecond which means that the deadline requirement was met and the protocol worked correctly during the experiment.

6.2.9 Non Real Time Traffic Experiment

RTXX Protocol allows us to use non real time and real time traffic on the same shared medium. The following experiment aims to measure the non real time traffic delays occurs on N_2 with the transmission schedule illustrated in *Figure 38.* A new thread is created on N_2 to generate non real traffic messages that contains the time instance of the system. Time instance in the message is captured at the packet creation time and sent by a container Ethernet packet. Another thread on N1 is also created to capture incoming Ethernet packets from N2. The following statements explains the operation of test software;

- The initialization process is started by 1588 PTP Time Synchronization process.
- After the time synchronization, the non real time packet generator thread on N_2 starts to send non real time Ethernet packets
- When a message arrives to N_1, the thread on N_1 parses the time instance value from the packet, calculates the time difference between its actual system time and the time instance coming from the Ethernet and stores it. After a specific number of samples collected test software finalizes itself.

Figure 40 illustrates the measured delay values for non real time traffic with unloaded system. The standard deviation for this experiment is too high because the scheduling for the test thread is non real time.

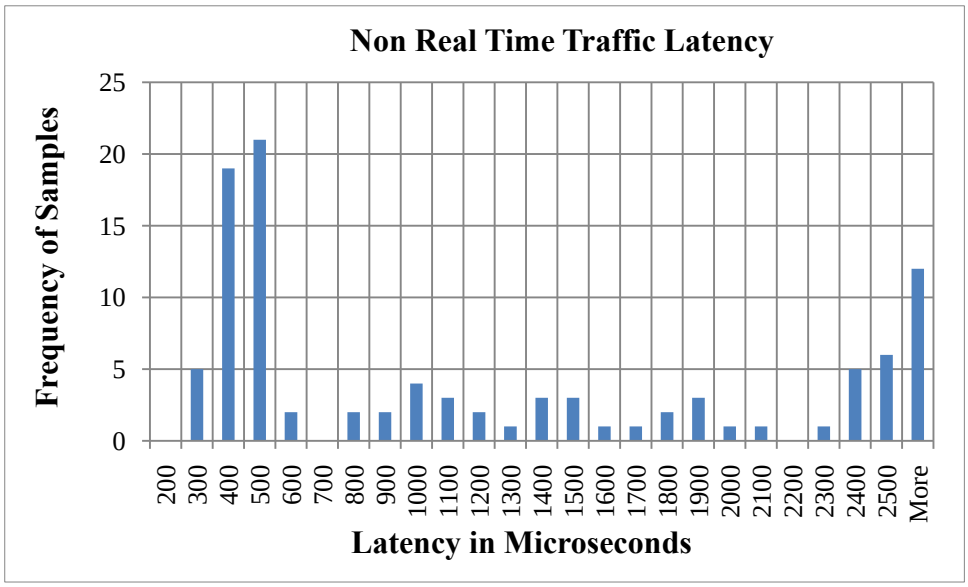

Figure 40: RTXX Latency for 4mS Deadline for Real Time Scheduling

Table 21: Experimental Results RTXX Latency for 4mS Deadline for Real Time Scheduling

Number of Samples	Minimum	Maximum
100	283	100023

CHAPTER 7

CONCLUSIONS

In this thesis performance of the Real Time Ethernet ,IEEE 1588 Precision Time Protocol and RTXX Protocol implementations are experimentally evaluated and the results of the experiments are presented.

Performance metrics for the Real Time Ethernet implementations are determined as periodic timer accuracy, periodic timer synchronization accuracy, slot switching latency. and one way packet transmission time. Results of the experimental measurements interpreted in *Section 6.2.1, Section 6.2.2 and Section 6.2.3* and *Section 6.2.4* As a summary of the experiments, delay bounds was understood to be due to the thread priority levels and real time scheduling of the operating system directly.

For the IEEE 1588 Precision Time Protocol the experimental performance parameter determined as the synchronization accuracy of the implementation. The experimental results are interpreted in *Section 6.2.5.* it is seen that the synchronization accuracy is depending on priority of the threads in the system. Nonetheless it is seen that packet size does not have remarkable effect on synchronization accuracy. Average accuracy for IEEE 1588 time synchronization is presented as 38 microseconds for 60 Byte packets with a standard deviation of 28,72,

In this thesis, the phase correction algorithm for IEEE 1588 Precision Time Protocol is not implemented due to the fact that both lab computers are exactly the same. For future studies, this feature can be implemented to current IEEE 1588 Precision Time Protocol and additional experiments can be done on different hardware architectures.

This thessis only implements the communicational operation of the RTXX Protocol with a limited request number per message. For fully functional RTXX implementation that explained in *Chapter 3*, a communication model should be defined and message creation algorithms based on this model should be provided

The system level tests underwent on the lab nodes. Performance bottlenecks determined as hardware clock source, priority of the application threads and real time scheduling capability of the Operating System that experiments run on.

Priority queue interaction between separate nodes is tested by the test software which indicates that both Real Time Ethernet Implementation and RTXX Protocol work seamlessly in the system.

REFERENCES

[1] J. Baillieul ve P.J. Antsaklis, "Control and Communication Challenges in Networked Real-Time Systems," Proceedings of the IEEE , vol.95, no.1, pp.9-28, Jan. 2007.

[2] J.R. Moyne ve D.M.Tilbury, "The Emergence of Industrial Control Networks for Manufacturing Control, Diagnostics, and Safety Data," Proceedings of the IEEE , vol.95, no.1, pp.29-47, Jan. 2007.

[3] J.-.Decotignie, "Ethernet-based real-time and industrial communications," Proceedings of the IEEE, vol. 93, no. 6, pp. 1102-1117, 2005.

[4] J.-.Decotignie, "The Many Faces of Industrial Ethernet [Past and present]," IEEE Industrial Electronics Magazine, vol. 3, no. 1, pp. 8 - 19, 2009.

[5] J. Thomesse, "Fieldbus technology in industrial automation," Proceedings of the IEEE, vol. 93, no. 6, pp. 1073-1101, 2005.

[6] John C. Eidson, Measurement, Control, and Communication Using IEEE 1588, Springer, 2006.

[7] K. Correll, N. Barendt, ve M. Branicky, “Design considerations for software only implementations of the IEEE 1588 precision time protocol”, Proc. Conference on IEEE-1588 Standard for a Precision Clock Synchronization Protocol for Networked Measurement and Control Systems, NIST and IEEE, 2005.

[8] J. C. Eidson ve K. Lee, “Sharing a common sense of time”, IEEE Instrumentation and Measurement Magazine, vol. 6, no. 1, pp. 26-32, 2003.

[9] 1st IFAC Workshop on Dependable Control of Discrete Event Systems, 2007.

[10] A. Avizienis, J.-C. Laprie, B. Randell, ve C. Landwehr, "Basic concepts and taxonomy of dependable and secure computing," IEEE Transactions on Dependable and Secure Computing, vol. 1, no. 1, pp. 11-33, 2004.

[11] M. Felser ve T. Sauter, "Standardization of industrial Ethernet - The next battlefield?," IEEE International Workshop on Factory Communication Systems, 2004.

[12] IEEE Std 802.3 Part 3: Carrier sense multiple access with collision detection (CSMA/CD) access method and physical layer specifications

[13] Timekeeping in VMware Virtual Machines, Infotmation Guide, http://www.foedus.com/downloads/solutions/whitepapers/vmware_timekeeping.pdf

[14] IA-PC HPET (High Precision Event Timers) Specification, October 2004

[15] http://www.osadl.org/Thomas-Gleixner.hannover-2008-thomas-gleixner.0.html

[16] http://en.wikipedia.org/wiki/POSIX

[17] J. Vidal, F. Gonzálvez, I. Ripoll. POSIX TIMERS implementation in RTLinux

[18] https://rt.wiki.kernel.org/index.php/RT_PREEMPT_HOWTO

[19] http://www.osadl.org/Latest-Stable-Quick-RT-Preempt-kerne.realtime-kernel-installation.0.html

[20] B. Thangaraju, Linux Signals for the Application Programmer, Mar, 2003

[21] He, Jialong, LINUX System Call Quick Reference

[22] Choudhary, Amit , Implementing a System Call on Linux 2.6 for i386, Revison 1.0, 2006, http://tldp.org/HOWTO/html_single/Implement-Sys-Call-Linux-2.6-i386/

[23] IEEE 1588 Standard for a Precision Clock Synchronization Protocol for Networked Measurement and Control Systems.[Online]. Available:http://ieee1588.nist.gov, 2002.

[24] http://www.home.agilent.com/agilent/home.jspx?cc=US&lc=eng

[25] http://www.ieee.org/index.html

[26] O'Farrell Patrick,Rosselot David, IEEE 1588 Synchronization Over Standard Networks Using the DP83640, 2009

[27] Tan, Alexander, Synchronizing Networks with IEEE 1588 Precision Time Protocol (PTP),

[28] http://en.wikipedia.org/wiki/Binary_heap

[29] http://www.ethermanage.com/ethernet/ch13-ora/ch13.html#81137

[30] K. Schmidt, E. Schmidt, and J. Zaddach, "RTDESC – A Real-time Communication Protocol for Distributed Discrete Event Control"

[31] J.-P. Thomesse, "Fieldbus technology in industrial automation," Proceedings of the IEEE, vol. 93, pp. 1073 – 1101, 2005.

[32] J.-D. Decotignie, "Ethernet-based real-time and industrial communications," Proceedings of the IEEE, vol. 93, pp. 1102 – 1117, 2005.

[33] M. Felser, "Real-time Ethernet - industry prospective," Proceedings of the IEEE, vol. 93, pp. 1118– 1129, 2005.

[34] P. Ramadge and W. Wonham, "The control of discrete event systems," Proceedings IEEE, Special Issue Discrete Event Dynamic Systems, vol. 77, pp. 81–98, 1989.

APPENDIX A

System Calls

System calls are the services provided by Linux kernel to attract kernel resource from a user space application [21].

A system call executes in the kernel space and a user program executes in the user space. In a Linux system, hardware access is restricted to the kernel space to protect the hardware routines from user space programs. Some cases, user space application requires to access directly to hardware to perform the specific job, like high precision timer access. In this case, there is a special need for bridging user programs to hardware which is called “system calls”.

System call implementations are dependent to microprocessor architecture. Every system call has a unique number associated with it. For Intel x86 architecture, when a user space program calls a system call, a library routine traps the kernel via executing the special “INT 0x80” assembly instruction and the associated number of the system call is passed to kernel via EAX register. The arguments of the system call are also passed to kernel via EBX,EBC, etc. register. Return value of the system call is passed from kernel to user program via other CPU registers. [22]

OpenSUSE 11.x is based Linux kernel version 2.6.27.7-9 so system call implementation method for RTXX protocol is described for this specific kernel version.

Modified kernel source files:

/usr/src/linux-version/include/linux/netdevice.h

/usr/src/linux-version/arch/x86/kernel/syscall_table_32.S

/usr/src/linux-version/include/asm-x86/unistd_32.h

/usr/src/linux-version/include/linux/syscalls.h

/usr/src/linux-version/Makefile

/usr/src/linux-version/include/linux/netdevice.h

netdevice.h is the core of network device structure in Linux. Every network interfaces in the system is based on net_device structure which is referenced in the netdevice.h file. To ensure that the locking mechanism can run on all interfaces, a special integer parameter is added in net_device structure. With this modification, every network interface may be locked independently from RTXX protocol application.

There are two integer arguments are implemented in the net_device structure. First one is MESSAGETYPE and the second one is POCKETMODE. MESSAGETYPE argument is used by the interface driver for determining the time slot. If the time slot belongs to real time traffic, driver locks its transmit queues coming from Layer 3 protocols, like IP, otherwise driver works as usual. POCKETMODE argument is used for shape the non real time Layer 3 traffic.

```
Struct net_device {
.
.
/* Modification in net_device structure */
int      MESSAGETYPE;
int      POCKETMODE;
.
.
}
```

Code 6: net_device Structure

/usr/src/linux-version/arch/x86/kernel/syscall_table_32.S

This file contains the system call names in the kernel. In order to provide full control of transmission on network interfaces in user space applications as RTXX Protocol

Application. Every new system call should be added to the end of system call list located in this file. There are three new system call added to the the sytem call list as below.

.long sys_lockRTXXDevice

.long sys_unlockRTXXDevice

.long sys_sendToRTXXDriver

lockRTXXDevice system call locks the transmission queues of the related interface. unlockRTXXDevice system call releases the transmission queues of the related interface. sendToRTXXDriver system call transfers user modified socked buffer directly to the interface's driver transmission routine without using the socket interface. This system can be used anytime by the user application.

/usr/src/linux-version/include/asm-x86/unistd_32.h

This file contains the system call numbers which is transferred to kernel through EAX register when the system call is invoked by the user application. New system call numbers should be defined in the file to notify the kernel about new system calls. Define parameters of the new system calls should be added to the end of predefined system call list. Highest system call number in the original file should increment by one and assigned to firstly added system call. For later arrivals, highest system call number should also be incremented by one and assigned as their system call number. If the highest system call number in the system is 332 then new system calls should be defined as below:

#define __NR_ lockRTXXDevice 333

#define __NR_ unlockRTXXDevice 334

#define __NR_ sendToRTXXDriver 335

In addition to newly added system call numbers, total system call number value should also be modified. If the highest system call number is 335, then total system call number should be assigned as 336 because system call number index starts from 0. If __NR_syscalls definition is not exist in the file, user should define it as;

#define __NR_syscalls 336

/usr/src/linux-version/include/linux/syscalls.h

This file contains the declaration of system calls. All the system call arguments should be declared in this file. Kernel will use these declarations for system call trap procedures. New system call declarations should be added at the end of the file as below;

asmlinkage long sys_lockRTXXDevice(char * device);

asmlinkage long sys_unlockRTXXDevice(char * device);

asmlinkage long sys_sendToRTXXDriver(char * device, struct sk_buff __usr *skb);

To notify compiler about socket buffer structure which is referenced in sendToRTXXDriver system call,

Struct sk_buff;

should be added at the beginning of the file.

/usr/src/linux-version/Makefile

Makefile of the linux kernel should be modified to add new system calls to kernel. In the Makefile, directory of the new system call function should be referenced to be compiled and linked to Linux kernel. A new folder, named "RTXXSystemCalls", is created in "/usr/src/linux-version/" directory and new system call source codes located in that folder. To reference that folder in the Makefile, user should search the " core-y +=" parameter and add the folder name at the end of the folder list.

Modified driver source code

Driver of the Ethernet interface should be modified to bring locking mechanism to RTXX Protocol implementation. To provide minimum impact on the driver source code, a new transmission function, "rtxx_start_xmit" , is added to source code instead of changing standard driver routines and the transmission entry function is routed to the new transmission routine. By this method, all the packets coming from operating system buffers can be controlled before reaching the real transmission function. Following changes have been made in the driver source code;

Adding a psudo transmission function:

A psudo transmission function is added to driver source to control realtime and non real time pocket traffic. Psudo code for the psudo transmission function is illustrated in *Code 7*

```
psudoTransmitFunc () {

        if (timeSlot==REALTIME){
                lockNetwork();
        }
        else{
                if (nonRtTrafficType==ONESHOT){
                        realTransmitFunc();
                        lockNetwork();
                }
                else {
                        realTransmitFunc();
                }
        }
}
```

Code 7: Psudo Transmission Function

Normally driver functions can not be accessed from kernel modules or user space function to avoid conflicts in the system, however RTXX protocol should access the driver codes to control the network traffic. To provide this capability to RTXX protocol applications, real transmission function of the network driver should be added to Linux kernel symbol table via EXPORT_SYMBOL() macro as below;

EXPORT_SYMBOL(realTransmitFunc);

In addition to that macro “static” identifier in the definition of realTransmitFuntion should be removed.

After defining the psudo transmit function and exporting it to the kernel symbol table, driver entry function should be redirected to this psudo function by modifying related part of the driver as below;

dev->hard_start_xmit = &psudoTransmitFunc;

New directories and files created for the Linux kernel:

/usr/src/linux-version/RTXXSystemCalls

/usr/src/linux-version/RTXXSystemCalls/lockRTXXDevice.c

/usr/src/linux-version/RTXXSystemCalls/unlockRTXXDevice.c

/usr/src/linux-version/RTXXSystemCalls/sendToRTXXDriver.c

/usr/src/linux-version/RTXXSystemCalls/Makefile

/usr/src/linux-version/RTXXSystemCalls

A new folder, named "RTXXSystemCalls", is created in "/usr/src/linux-version/" directory to put new system call source codes in it.

/usr/src/linux-version/RTXXSystemCalls/lockRTXXDevice.c

A new file created as the source code of lockRTXXDriver system call

/usr/src/linux-version/RTXXSystemCalls/unlockRTXXDevice.c

A new file created as the source code of unlockRTXXDriver system call

/usr/src/linux-version/RTXXSystemCalls/sendToRTXXDriver.c

A new file created as the source code of sendToRTXXDriver system call

/usr/src/linux-version/RTXXSystemCalls/Makefile

A new file created as the Makefile for the system calls. Makefile of the linux kernel references this make file to build and link new system calls. To notify compiler about systemcall source files following parameters should be added to this Makefile;

obj-y: lockRTXXDevice.o unlockRTXXDevice.o sendToRTXXDriver.o

APPENDIX B

Configuring and Building Linux Kernel for RTXX Implementation:

Linux kernel must be rebuild to activate the modifications on kernel and driver source codes. Building process of the kernel is explained in the following parts:

Configuring Linux Kernel

There are several ways to configure Linux kernel. Most user friendly method for the configuration is doing it by graphically. Before starting the configuration, user must login the command prompt with administrator/root permissions .Then to configure Linux kernel , user should enter the kernel source directiory, "/usr/src/linux-version/" and enter following command to command prompt;

- make menuconfig

Graphic output of the kernel configurator illustrated in *Figure 41*

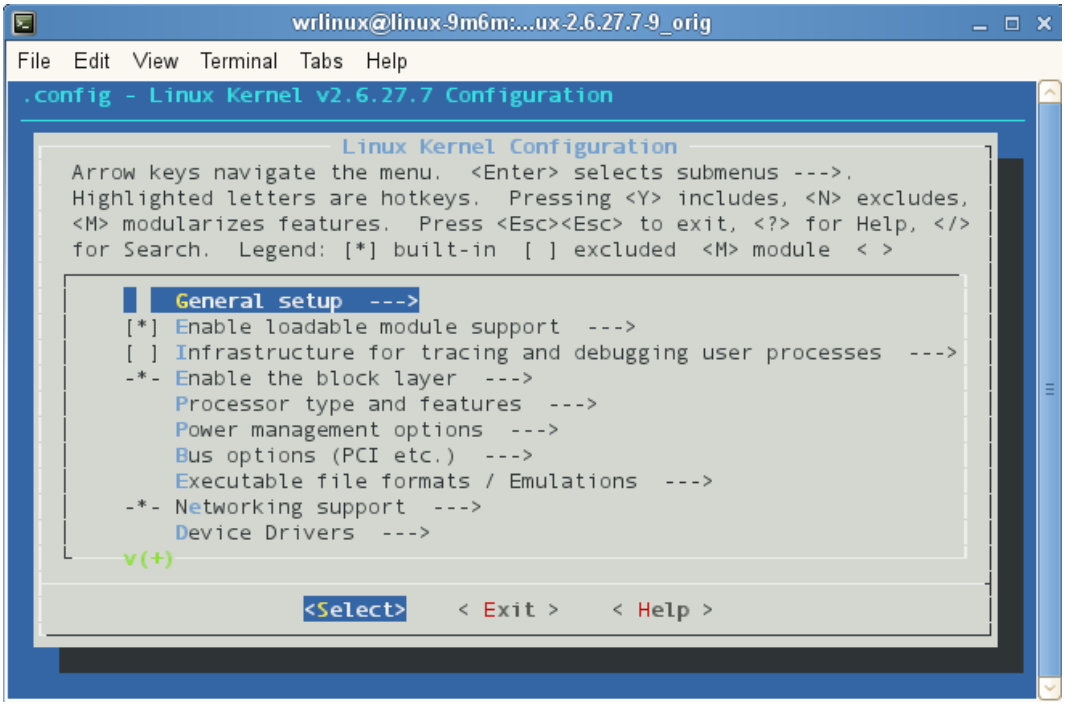

Figure 41: Linux Menu Config Window

Standard linux kernels are designed for general purpose requirements like multithreading, multimedia, graphical applications etc. Because of this, the following changes illustrated in *Table 22* must be done to Linux kernel to provide it more real time characteristics.

```
Processor type and features
  (x) Tickless System (Dynamic Ticks
  (x) High Resolution Timer Support
  (x) HPET Timer Suppor
  Preemption model
    (x) Preemptible Kernel (Low-Latency Desktop
  Timer frequency
    (x) Timer frequency (1000)
```

Table 22:Menuconfig Configuration

Building kernel

After configuration process, linux kernel should be recompiled for new kernel. . Before starting the configuration, user must login the command prompt with administrator/root permissions .Then to compile Linux kernel , user should enter the

kernel source directiory, “/usr/src/linux-version/” and enter following command to command prompt;

- make

Building process should start immediately. Although the estimated compilation time is about 1 hour, this time may vary depending on CPU speed. After building the kernel, kernel modules should be installed the related directories. This process is done by the following command;

- make modules_install

Final step is generating the final Linux image and installing it to the boot location. This process is done by the following command

-make install

Computer should be restarted to boot the latest kernel.