IMPLEMENTATION AND EVALUATION OF A SYNCHRONOUS TIME-SLOTTED MEDIUM ACCESS PROTOCOL FOR NETWORKED INDUSTRIAL EMBEDDED SYSTEMS

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

BY

AHMET KORHAN GÖZCÜ

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

SEPTEMBER 2011

Approval of the thesis:

IMPLEMENTATION AND EVALUATION OF A SYNCHRONOUS TIME-SLOTTED MEDIUM ACCESS PROTOCOL FOR NETWORKED INDUSTRIAL EMBEDDED SYSTEMS

submitted by **AHMET KORHAN GÖZCÜ** 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** _____

Assist. Prof. Dr. Şenan Ece Schmidt
Supervisor, **Electrical and Electronics Engineering Department,METU** _____

Examining Committee Members:

Prof.Dr. Semih Bilgen
METU _____

Assist. Prof. Dr. Şenan Ece Schmidt
METU _____

Prof. Dr. Gözde Bozdağı Akar
METU _____

Assoc. Prof. Dr. Cüneyt Bazlamaçcı
METU _____

Ms. Sc. Bora Kartal
ASELSAN _____

Date: _____

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: AHMET KORHAN GÖZCÜ

Signature :

ABSTRACT

IMPLEMENTATION AND EVALUATION OF A SYNCHRONOUS TIME-SLOTTED MEDIUM ACCESS PROTOCOL FOR NETWORKED INDUSTRIAL EMBEDDED SYSTEMS

Gözcü, Ahmet Korhan
M.Sc., Department of Electrical and Electronics Engineering
Supervisor : Assist. Prof. Dr. Şenan Ece Schmidt

September 2011, 74 pages

Dynamic Distributed Dependable Real-time Industrial communication Protocol family (D^3RIP), has been proposed in the literature considering the periodic or event-based traffic characteristics of the industrial communication networks. In particular, D^3RIP is designed to dynamically adapt the bandwidth allocation for distributed controller nodes depending on the instantaneous communication requirements.

D^3RIP framework consists of two protocol families: Interface Layer (IL) protocol family, which is responsible for providing the accurate time-division multiple access (TDMA) on top of a shared-medium broadcast channel, and Coordination Layer (CL), which is defined to fulfill the external requirements of IL. In this thesis, the hardware adaptations of the two protocols, Real-time Access Interface Layer (RAIL) and Time-slotted Interface Layer (TSIL), of the IL protocol family, are implemented. Their performance on both personal computers (PC) and development kits (DK) are observed.

Keywords: industrial networks, real-time communication, Ethernet

ÖZ

ŞEBEKELENMİŞ ENDÜSTRİYEL GÖMÜLÜ SİSTEMLER İÇİN SENKRON ZAMAN OLUKLU ORTAMA ERİŞİM PROTOKOLÜ GERÇEKLENMESİ VE DEĞERLENDİRİLMESİ

Gözcü, Ahmet Korhan
Yüksek Lisans, Elektrik-Elektronik Mühendisliği Bölümü
Tez Yöneticisi : Yar. Doç. Dr. Şenan Ece Schmidt

Eylül 2011, 74 sayfa

Literatürdeki Dinamik Dağıtılmış Güvenilir Gerçek Zamanlı Endüstriyel İletişim Protokolü ailesi (D^3RIP), endüstriyel iletişim ağlarının periyodik ya da olay tabanlı olan trafik özellikleri göz önüne alınarak tasarlanmıştır. Özellikle, D^3RIP, anlık iletişim gereksinimlerine bağlı olarak, dağıtılmış denetleyici düğümler için dinamik olarak bant genişliği tahsisini adapte edecek şekilde tasarlanmıştır.

D^3RIP sistemi iki protokol ailesinden oluşur: paylaşılan bir orta yayın kanalının üstünde doğru çalışan zaman bölmeli çoklu erişimin (TDMA) sağlanmasından sorumlu olan Arayüz Katmanı (IL) protokol ailesi, ve IL'in dış gereklerini yerine getirmek için tanımlanmış olan Koordinasyon Katmanı (CL). Bu tez çalışmasında, IL protokol ailesinin iki protokolü olan, Gerçek Zamanlı Erişim Arabirim Katmanı (RAIL) ve Zaman Oluklu Arayüz Katmanının (TSIL), donanım üzerinde uyarlamaları gerçeklenecektir. Bunların hem kişisel bilgisayarlar (PC) hem de geliştirme kitleri (DK) üzerindeki performansları gözlenecektir.

Anahtar Kelimeler: endüstriyel ağlar, gerçek zamanlı haberleşme, Ethernet

To My Family

ACKNOWLEDGEMENTS

First of all, I thank my thesis supervisor Assist. Prof. Dr. Şenan Ece Schmidt and his husband Assist. Prof. Dr. Klaus Schmidt for their valuable critics, excellent supervision and support. Besides, I like to thank them providing me the opportunity to work on such an important issue. I also like to thank Ulaş Turan for his contributions on the integration of our thesis.

I am in great debt to my mother Hayriye Gözcü and father İbrahim Gözcü for their patience and supports. Without them and their praying, I am sure that this thesis would not exist. In addition I would like to thank specially to my sister Nilüfer Sakancı and brother Mustafa Kaan Gözcü. Even if they could not be with me all the time, I can feel their best-wishes and prays. Therefore, I would like to give my biggest appreciation to my family. Also I would like to send my thanks to all my relatives, supporting me.

I would like to thank my bro Mahmut Uğur Eren, for his invaluable friendship and being my oldest and closest friend. I would like to express my thanks to all my friends for their support and fellowship. I would like to thank my friends from high-school separately. İrem Şafak, Aslı Selver, Mete Kaymak, Mehmet Altuğ Şahin, Ahmet Gökhan Coşkun and Bilge Bakın, bring joy and happiness to me always. They become a source of moral and motivation for me, especially during this thesis.

I wish to thank ASELSAN A.Ş. for giving me the opportunity of continuing my education. I would like to express my special appreciation to my colleagues and seniors in HBT-ETD-TYM, for their contribution on the improvement of my engineering skills, which makes it possible to finish this thesis. Also I am very grateful for their patient audience to my complaining in the mornings, especially for the last months of my thesis.

I would like to give a special thanks to Ahmet Emrah Demircan for his contributions for the frame segmentation part of my thesis, and helping me when I stuck to problems in Linux.

I would also like to thank TÜBİTAK-BİDEB for their financial support during my graduate education.

TABLE OF CONTENTS

LIST OF TABLES

TABLES

LIST OF FIGURES

FIGURES

LIST OF ABBREVIATIONS AND ACRONYMS

API	Application Programming Interface
CAN	Controller Area Network
CL	Coordination Layer
CT	Collision Threshold
D^3RIP	Dynamic Distributed Dependable Real-time Industrial communication Protocol family
DART	Dynamic Allocation Real-time Protocol
DK	Development Kit
EIP	Ethernet/IP
EPA	Ethernet for Plant Automation
EPL	PowerLink
FPGA	Field Programmable Gate Array
IL	Interface Layer
MAC	Medium Access Control
nRT	non-Real Time
OSADL	Open Source Automation Development Labs

OSI	Open System Interconnection
PC	Personal Computer
RT	Real Time
SM	Shared Medium
SMP	Symmetric Multi-Processing
TCNet	Time Critical Control Network
TCTL	Transmit ConTroL register
TDMA	Time Division Multiple Access
TSIL	Time Slotted Interface Layer
URT	Urgency-based Real-time Protocol

CHAPTER 1

INTRODUCTION

As the technology improves, industrial control applications become increasingly complex and large-scale. The data volumes to be transmitted on the industrial networks are increasing rapidly, which causes a demand for faster communication techniques. Traditional industrial communication techniques, i.e., proprietary fieldbuses such as Controller Area Network (CAN), LonWorks and Profibus, could not satisfy these needs of the industry, because of their high cost, low speed, incompatibility with different data bus technologies, and unadaptable structure for improvements.

Ethernet becomes the primary alternate for the replacement of the fieldbuses, which are inadequate to meet the demands of the industry. Having low cost and high speed, being widely used and successful; put Ethernet ahead of other candidates. Thus, nowadays many of the industrial control devices are being manufactured with Ethernet network interface cards.

Apart from these advantages, Ethernet has a major deficiency, which is being nondeterministic. This randomness is caused by the truncated binary exponential backoff algorithm [1], which is used when a collision occurs. According to this algorithm, the nodes, sending the messages that collide, wait an amount of time chosen randomly from a definite interval. Thus, it is impossible to calculate an upper bound for the waiting time before a message is successfully transmitted from one station to another. It is only possible to derive the probability that this waiting time will exceed a given value, which is not adequate for control applications that require hard bounds on the waiting time.

To overcome this problem of Ethernet, many solution techniques are proposed in the literature.

Some techniques like Ethercat [2], SERCOS III [3] and ProfiNet [4], suggest altering the

MAC layer by adapting Ethernet network interface hardware. These types of solutions carry the disadvantages of the fieldbuses like high cost hardware needs and being incompatible with similar data busses.

MODBUS RTPS [5] and PROFINET SRT [4], on the other hand, aims to decrease the collision probability and reaction times, which only improve the performance and could not provide real-time (RT) guarantees.

To prevent collisions, using switches is another type of solution, as proposed in Ethernet/Ip (EIP) [4]. However, it leads to other problems such as queuing delays and message loss because of the limited queue size.

Final and most preferred technique to get rid of the randomness of Ethernet is adding an additional layer, that prevents simultaneous access to the shared medium (SM) and thus avoids collisions, on top of the MAC layer. This additional layer could use master-slave (Powerlink (EPL) [4]), token passing (Time Critical Control Network (TCNet) [4]), or static Time Division Multiple Access (TDMA) (Ethernet for Plant Automation (EPA) [4]) solutions. Each of these techniques carries different disadvantages. Master-slave solutions have single point of failure, undistributed structure and low efficiency. Token passing solutions lack dependability, especially in the case of losing the token. Static TDMA solutions have low efficiency due to the guard periods and error recovery precautions.

The major reason for the inefficiency of the techniques mentioned is that they ignore the special requirements of the control-purpose applications by simply modifying the solutions used for home and office networks. Unlike home and office networks, which have random traffic, the messages in the industrial communication networks are generated in either periodic or event-based, i.e., sporadic manner. Besides, in industrial control systems, the communication requirements of the system components are known at any moment, since automation control applications work in a deterministic way. However, the solutions in the literature make worst-case assumption, such as all messages sent at the same time, to meet the needs of hard real-time industrial control system applications, and capacity allocation is done accordingly. For this reason, the capacity allocation is inefficient.

Dynamic Distributed Dependable Real-time Industrial communication Protocol family (D^3RIP), on the other hand, has been developed in [6] considering these characteristics of the industrial

communication networks. Thus it proposes a TDMA solution with dynamic slot assignment according to the communication needs of the system components, which are known at any moment.

In this thesis, the protocols in the Interface Layer (IL) protocol family of D^{3}RIP framework, which are Real-time Access Interface Layer (RAIL) and Time-slotted Interface Layer (TSIL), are implemented on real hardware. The IL is responsible for providing the accurate TDMA on top of a shared-medium broadcast channel.

During the implementation of IL, seven challenges are faced. Four of them are already defined in [6], but needed to be detailed.

- minimizing the guard periods in the time slots
- frame fragmentation and reassembly
- interfaces with Shared Medium (SM)
- interfaces with Coordination Layer (CL)

The other three challenges are not mentioned in [6]. They occur as implementation requirements.

- obtaining synchronization over TDMA structure
- prioritizing synchronization packets over nRT packets
- starting the TDMA structure simultaneously

By overcoming these challenges, this thesis contribute to the detailed description of the D^{3}RIP framework.

The rest of the thesis is organized as follows. In Chapter 2, first, industrial network is described in terms of its communication needs and differences from home and office networks. By explaining the traditional fieldbuses' inefficiency, the Ethernet based solution techniques are inspected. Finally, the D^{3}RIP, which is a new synchronous time-slotted medium access solution that is proposed in [6], in the light of the deficiencies of the Ethernet based solutions in the literature is mentioned.

In Chapter 3, the challenges in the implementation of the protocols of the IL protocol family of D^3RIP framework are described. Besides some performance metrics that are used to measure the performance of these protocols are explained.

Chapter 4, describes the implementation of these protocols on real hardware by explaining how the challenges described in the Chapter 3 are overcame.

In Chapter 5, the performance of these implementations is analyzed in terms of the performance metrics defined in Chapter 3.

Finally, in Chapter 6, discussions and conclusions are presented. In addition, some future works are suggested.

CHAPTER 2

RELATED WORK

2.1 Industrial Communication Networks

Unlike home and office networks, which pay attention mostly to throughput and operate as a best effort delivery, industrial communication networks have deterministic behavior, due to the deterministic operation of the control applications. In industrial areas, networks are composed of different components with different communication needs; such as, sensors, actuators, controllers, supervisory stations, remote controllers, and diagnostic viewers.

Sensors and actuators are among the components of the industrial networks that need least intelligence. Sensors continuously gather data to the controller and it operates actuators according to these data. Thus, they work in a predefined periodic manner. Therefore, the communication between sensors-controllers and controllers-actuators need periodic messages with deadlines.

Supervisory stations, on the other hand, are the most intelligent part of the industrial communication systems. Supervisory station refers to the servers and software responsible for the hierarchical communication of the systems that are in hierarchically different levels, with the purpose of coordination. They mostly use event-based messages that need deterministic response time. Still, the messages to be used can be known earlier according to the state of the system, by using the systems dynamic model.

Apart from these messages with RT needs, there are also non-Real-time (nRT) messages in the industrial communication networks. These are needed by the remote controllers, which are responsible for the remote management of the system and diagnostic viewers, which are used for the monitoring purposes. These nRT messages are also event-based, like supervisory

control messages.

Control systems with distributed components described above are used since 1980's in industrial area. To accomplish the requirements of the messages described, initially all of the components were connected to each other by wires. This communication technique gives its place to industrial communication networks, because of its high cost of wires and difficulty of adding a new component.

The first industrial communication networks are proprietary fieldbuses such as CAN, LonWorks and Profibus [7]. These techniques seem not bringing long-term solutions as their high cost hardware needs, being slow, hard to improve and incompatible with similar data busses.

2.2 Industrial Ethernet

As a result of the disadvantages of the fieldbuses that mentioned in the previous chapter, they are replaced by Ethernet-based networks such as Profinet [4] and Ethernet Power Link (EPL) [8]. The main reason behind using Ethernet as a solution is its widely and successive usage. Since Ethernet is a widely used communication technique used in home and office networks, it draws attention of both academic and industrial researchers. Therefore, many improvements are made in the Ethernet which results in very cheap and fast communication, when compared with the fieldbuses.

Apart from these advantages of Ethernet, it has a major deficiency that is its nondeterministic behavior. The reason of this nondeterministic behavior is as a result of the truncated binary exponential backoff algorithm used for retransmitting a packet after a collision, which occurs when more than one node tries to transmit a message to a shared medium. According to the truncated binary exponential backoff algorithm [1], each node exposed to collision retransmits its message after waiting an amount of time chosen randomly from a definite interval. After each collision that the same message encounters, the upper limit for this wait duration doubles. This random waiting process after a collision causes Ethernet's nondeterministic behavior.

To overcome this problem of Ethernet and provide RT characteristics to it, different approaches are developed.

One technique is altering the Medium Access Control (MAC) layer by adapting Ethernet

network interface hardware. In this way, nondeterministic message transmission function of the Ethernet can be modified, as in Ethercat [2], SERCOS III [3] and ProfiNet [4]. Since this technique needs modification on the hardware, it carries the disadvantages of the proprietary fieldbuses, such as, high cost hardware needs and being incompatible with similar data busses.

Another technique is improving the performance of the Ethernet by decreasing the collision probability and reaction times. MODBUS RTPS [5] and PROFINET SRT [4] are some examples using this technique. Disadvantage of this technique is that it only improves performance to obtain near RT characteristics. Thus it could not provide hard RT guarantees.

Using switches is another technique that removes collision probabilities. Since switches put all the incoming messages to the queue before transmitting to other nodes. This controlled transmission prevents collision; however, it leads other problems such as queuing delays and message loss because of the limited queue size. Ethernet/Ip (EIP) [4] is among the protocols that use this technique.

Probably the most popular technique to bring RT characteristics to the Ethernet is adding an additional layer on top of the MAC layer. This additional layer avoids collisions by providing deterministic access to the shared medium. The most used solution types using this additional layer are master-slave solutions, token passing solutions, and TDMA solutions.

In master-slave solutions, a dedicated master is in the control of the communication. It continuously enquiry each slave, and if the slave has a message to send, it will respond to the enquiry with its message. The dummy enquiries, even in the absence of message to be transmitted, cause the efficiency of the protocol to drastically fall down. For example, the efficiency of the Powerlink (EPL) [4], which uses this type of solution, is calculated as 25% in [9]. Having single point of failure is another disadvantage of the master-slave type solutions.

In token passing solutions, a signal called a token is passed between the nodes. The node, which takes this token, is authorized to communicate. Thus, only one node is allowed to transmit at a time, which provides collision avoidance. Time Critical Control Network (TC-Net) [4] can be given as an example that uses token passing as a collision avoidance solution in Ethernet based industrial systems. One major disadvantage of token passing is that when a failure occurs, causing the token to be lost, the communication will stop.

In TDMA solutions, time is divided into time-slots and each node in the system is assigned

to a set of these slots. This unique assignment of time slots avoids collisions. This fully deterministic structure of TDMA provides robustness. On the other hand, this determinism can be quickly lost in the presence of errors and that the solutions are highly inefficient. For example, in [10], it is shown that a maximum of only 4% of the bandwidth is usable to carry useful traffic, when implementing a TDMA on switched gigabit Ethernet hardware. Also it explains that the major reason of this inefficiency is the Ethernet switches.

Ethernet for Plant Automation (EPA) [4], is an example using TDMA technique to bring RT characteristics to Ethernet. This protocol uses two phase TDMA structure. In the first phase, time-slots are statically assigned to the RT messages of each node. These slots belong to the node even if it has no message to send. In the second phase, which is used for nRT communication, the time-slots are assigned to the nodes according to their announcement, which is done at the end of the RT messages in the first phase. The static assignment of time slots to the nodes forces each node to transmit its synchronous traffic at the same frequency, which in turn reduces the efficiency and limits the applicability.

2.3 D^{3}RIP

As explained in the section 2.1, industrial communication networks do not have a random traffic. Instead, the messages in the industrial area are generated in either periodic or event-based, i.e., sporadic manner. However, the industrial network protocols in the literature ignore this situation. They are developed by only modifying the solutions used for home and office networks according to the needs of the applications. Therefore, these approaches use network capacity inefficiently.

D^{3}RIP framework proposed in [6], on the other hand, has arised to fill the gap between the industrial needs and the academic solutions. This framework and the protocols in it, consider the information from the control application, which is available via the application protocol interface, to dynamically know about the communication requirements of nodes and the timing of the messages. To provide a fully distributed structure, which provides robustness by preventing having a single point of failure, this information will be broadcasted on the shared medium and coordinated over all nodes on the network. Thus, the knowledge about the deterministic system behavior will be available to all nodes, which makes it possible to provide

RT guarantees and dependability, to compute the required maximum network bandwidth and to dynamically allocate the capacity among the nodes based on the communication requirements. In addition, the knowledge about the communication requirements allows using the remaining capacity after sending the RT messages for the nRT traffic.

As knowing about the communication requirements of the nodes and the timing of the messages, D^3RIP can schedule the RT messages. Thus, TDMA, which is more appropriate than master-slave and token passing, is chosen to bring RT characteristics to Ethernet. Since this scheduling is dynamically updated via the knowledge from the control application, the slot assignment in the TDMA structure is made in such a way to adopt these changes. This dynamic assignment of slots for RT messages and not using switches will overcome the inefficiency problem of the TDMA, mentioned in section 2.2.

CHAPTER 3

PERFORMANCE METRICS and IMPLEMENTATION CHALLENGES

3.1 Problem

D^3RIP framework consists of two protocol layers; Interface Layer (IL) and Coordination Layer (CL). IL is responsible for providing the accurate TDMA structure on top of a shared-medium broadcast channel, whereas CL is defined to fulfill the external requirements of IL. In this thesis, the hardware adaptations of the two protocols of the IL protocol family, which are RAIL and TSIL, are implemented. The implementation of these protocols are on both personal computers (PC) and development kits (DK).

3.2 Performance Metrics

After the hardware implementation of the RAIL and TSIL protocols on both PC and DK, their performance are measured. The numerical performance metrics to be used for this measurement are given and described below:

Throughput: This corresponds to the amount of RT and nRT traffic that can be carried over the network per unit time (in bps, frames/sec etc). For the computation of this metric, the minimum slot duration, including the guard periods, is calculated. Then the time needed for a message with maximum length to be transmitted on the shared medium is calculated. The ratio of the message's transmission time to the total slot duration gives the throughput of the implementation.

Maximum RT Throughput: Since D^{3}RIP framework is an industrial communication network, it focus on RT packet transmission. Still, some of the bandwidth should be allocated to the synchronization packets. To calculate this metric, first, the minimum number of slots needed for accurate synchronization is calculated. By multiplying the ratio of these slots to all slots, with the throughput, and subtracting the result from the throughput, the maximum RT throughput of the implementation is obtained.

Efficiency: This corresponds to the ratio of the effective throughput for RT packets to the total throughput of the shared medium. It is calculated by simply dividing maximum RT throughput to the exact throughput of the transmission line used.

Minimum RT Message Deadline: The most important need for the RT messages in the industrial communication networks is being able to put an upper bound for the time needed for the transmission. As D^{3}RIP framework assigns the slots dynamically according to the needs of the control application, when an urgent message is ready, it will be transmitted with the next RT slot. Still, according to the scheduling, this message will wait during the nRT slots before a RT slot came. Therefore, to calculate this metric, first the maximum wait time for a RT slot is calculated. By adding the time needed for the transmission of the message to the CL of the receiver nodes ($dSlot - rem$) to this time, achievable minimum RT message deadline is calculated.

3.3 Challenges

During the implementation of the protocols in the IL protocol family of D^{3}RIP framework, six major challenges are faced. The definitions of these challenges are defined below.

Synchronization over TDMA Structure The most important requirement of running a correct TDMA structure is the accurate synchronization of the clocks on the nodes in the system. In general, synchronization is implemented on a separate line other than the one uses TDMA structure. However, this is not valid in D^{3}RIP framework case, since it is defined as a generic solution working on a shared medium only. Therefore, the solution described in subsection 4.3.1 is proposed.

Prioritizing Synchronization Packets over nRT Packets In implementations that are using nRT synchronization application, if the nRT traffic is very high, the synchronization

messages could be timed out. Therefore, the synchronization packets should have priority over other nRT packets. The prioritizing of the IEEE1588 synchronization packets is realized as described in subsection 4.3.2.

Minimizing the Guard Periods in the Time Slots To have a good performance from a TDMA structure, the guard periods in the time slots should be as small as possible. To minimize the guard periods, the accurate timing of the actions defined in the IL protocol (Sec.4.2) is crucial. The studies on obtaining accurate timings are described in subsection 4.3.3.

Starting the TDMA Structure Another important necessity of operating an accurate TDMA structure is to start it on all the nodes in the system simultaneously. Thus, each node will number the time slots same. The works on the simultaneous start of TDMA structure can be found in subsection 4.3.4.

Frame Fragmentation and Reassembly In D^3RIP framework, the slot sizes are defined constant. The size of the slots can be chosen according to the longest RT message which is known in each application. Still, the nRT packets coming from the upper layer applications can be greater than the longest RT message. Therefore; long nRT packets should be divided into frames of appropriate size that can be reassembled at the destination. The designs of the fragmenter and the reassembler are given at Sec.4.4.

Interface with Shared Medium (SM) For the communication between SM and IL, transmit and receive functions should be defined. These functions should be as close to the shared medium as possible, in order to transmit and receive with minimum delay, which will increase the performance of the TDMA structure. The proposed solution for this challenge is described in Sec.4.5.

Interface with CL Another interface of IL is between CL. The needs of this interface is signalling for an action and transferring data. The speed of the techniques to be used for signalling and data transferring also affects the performance of D^3RIP framework. The studies on these techniques are given in Sec.4.6.

CHAPTER 4

D^{3}RIP's IL IMPLEMENTATION

4.1 Detailed Description of the D^{3}RIP

Dynamic Distributed Dependable Real-time Industrial communication Protocol family, D^{3}RIP, defines a new real-time communication framework. The main difference of this framework from the previous ones is that it uses the information that comes from the control applications to arrange the scheduling of the messages. By doing so, D^{3}RIP adapts the communication network according to the needs of the control applications.

Another important feature of this framework is its completely distributed structure. All messages in the system are broadcasted and so, all communication nodes of the framework keep exactly the same scheduling. This distributed structure of D^{3}RIP makes it possible to provide real-time guarantees and dependability by letting all nodes know about the deterministic system behavior. Besides, making the framework completely distributed increases the dependability of the system since it avoids having a single point of failure as opposed to relying on the correct functionality of a dedicated master node [11].

D3RIP framework defines two protocol families, Interface Layer (IL) and Coordination Layer (CL), to communicate over a Shared Medium (SM) (Fig. 4.1). IL is responsible for putting up the TDMA structure by adjusting the timings of the slots. CL, on the other hand, by using the correct time slotted operation served by IL, dynamically allocates bandwidth to the RT applications, according to their needs.

SM, in the D^{3}RIP framework, is a generic broadcast channel for shared-medium networks, like Ethernet. It transmits the message m that comes from any of the connected device's IL via the IL2SM(m) and broadcasts it to all connected ILs via the SM2IL(m). (Fig. 4.2) Message

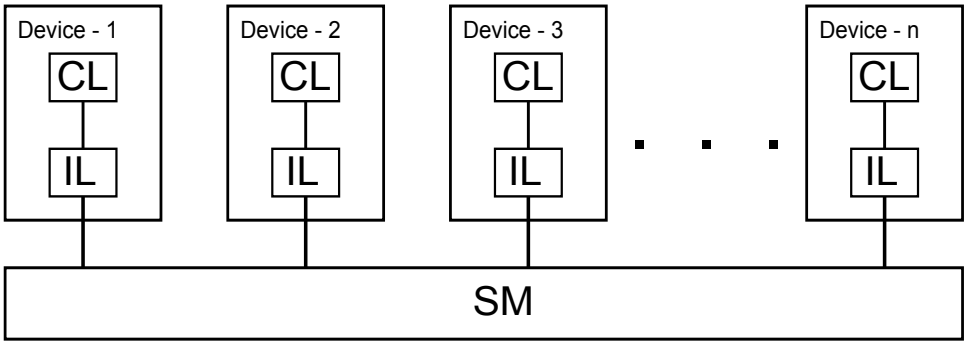

Figure 4.1: Overview of D^{3}RIP Framework

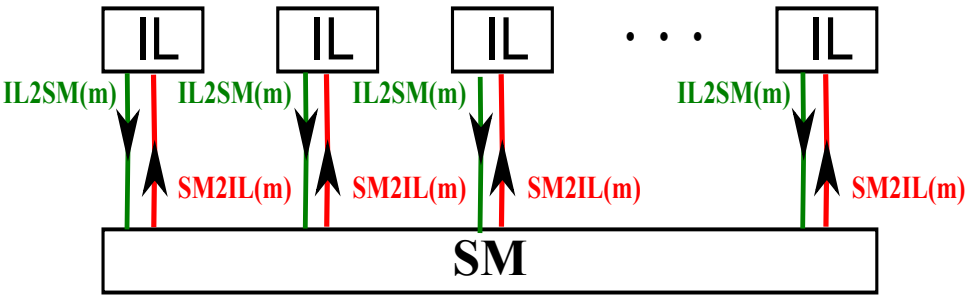

Figure 4.2: Message Transmission between IL and SM

transmission starts with the IL2SM(m) action, which passes the message m from IL to SM. After this action occurs from any of the connected devices, the SM will be busy until the transmission of the message m finishes. During this period none of the IL's connected should try to transmit another message to avoid collision, which is a stop condition for SM as can be seen in Appendix A.1. Collision avoidance is in the scope of IL, which provides the time-slotted operation.

Interface Layer (IL), which stays between SM and CL, provides a correct time-slotted operation by assigning time slots to unique devices in the system. It also decides the type of the slot, i.e., if the slot is reserved to transmit an RT message or an nRT message. The determination of the type of the slots and their assignments to unique nodes is done by processing both the local information in the IL and the arguments passed from the CL as a response to the request from IL. Besides time-slotted operation, another task of IL is passing messages from CL and nRT applications to SM and vice versa. While passing messages to SM, IL is responsible for the division of the big messages into sub-frames in order to fit them into the transmission

window. IL should of course combine these sub-frames on the receiver sides before sending them to the upper layers.

Further details of the IL protocol family will be discussed in the next section. Besides, RAIL and TSIL, which are the two protocols in this protocol family, will be detailed. Therefore; that much information is enough to understand a general overview of the D^{3}RIP framework.

The last part of the D^{3}RIP framework is the Coordination Layer (CL) protocol family. CL's basic duty is to guarantee the real-time needs of the connected applications by using the time-slotted medium access offered by IL. To fulfill these RT requirements, CL processes and broadcasts RT messages, which includes both the data that comes from the RT applications and the parameters related to CL protocol. CL uses these parameters to arrange the scheduling information of the time-slots. CL arranges the scheduling according to the deadline constraints of the messages of the RT applications.

Whenever a request is issued by IL, CL returns with information about the ownership and the type of the slot, i.e., if it is RT or not. CL also puts a boundary to the delay between IL's request and CL's response, in order to guarantee the protocol specific requirements.

Under the Coordination Layer protocol family, two different protocols are defined. The first one is the Dynamic Allocation Real-time Protocol (DART), which assigns allocated RT slots to specific controller devices and dynamically updates the ownership of the slots depending on the state of the control application. The other one is the Urgency-Based Real-time Protocol (URT), which uses a priority queue to dynamically update the right to transmit of each device considering their communication requests. Since CL implementation is out of the scope of this thesis, no further information is given, except the TIOA description in Appendix A.3

4.2 Detailed Description of the IL

As briefly explained in the previous section, IL's main task is transmitting both RT messages that come from CL and nRT messages that come from the non-Real-time applications, in a collision avoiding time-slotted behavior. In the D^{3}RIP framework, IL protocol family is defined explicitly in terms of input, output, and internal actions and variables.

The variables of IL protocol family consists of the protocol related variables; now, next, `myIL`,

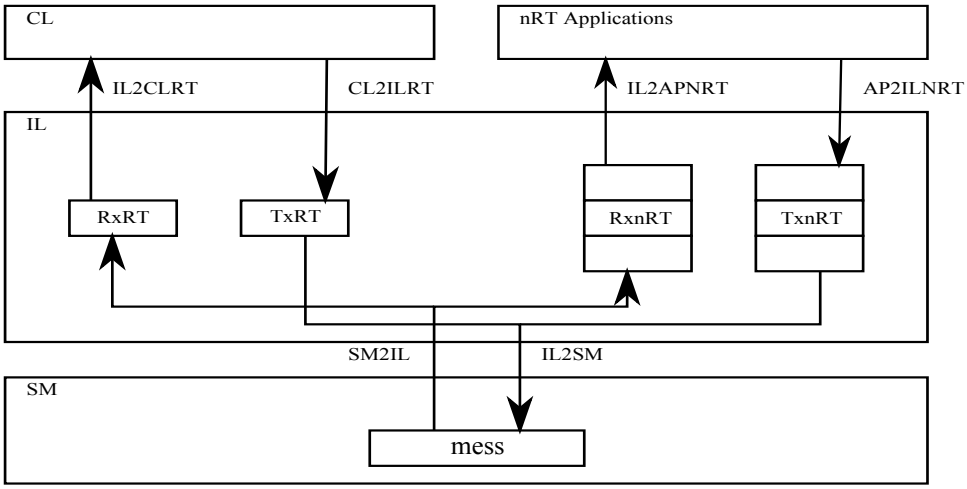

Figure 4.3: Message Transmission of IL Protocol Family

vIL, and the variables used to store messages; TxRT, RxRT, TxnRT, RxnRT. For storing RT messages, buffers with one message length (TxRT and RxRT) are used, whereas for nRT messages, FIFO-queues (TxnRT and RxnRT) are used. (Fig. 4.3) The reason of this choice is that RT messages are transmitted one at a time in a well-defined behavior, as explained later in this section. nRT messages, on the other hand, are sent and received by the upper layer applications at any time, unaware of the IL protocol.

The two of the protocol related variables, now and next are timing variables. The only real variable, now, shows the current time and used to activate the actions in each time slot. (Fig. 4.4) The integer variable, next, on the other hand, holds the end of the current time-slot, which is also the beginning of the next one.

The other two variables, myIL and RTIL are used to determine the properties of the time-slot. RTIL, indicates if the slot will be used for RT communication or nRT communication. On the other hand, myIL holds the ownership of the slot, i.e., if it will be used by this device or it is owned by another one.

The last variable defined in the IL protocol family, is the protocol specific variable vIL. It is a structure containing additional information about the scheduling of the time slots. Since the contents of vIL changes according to the chosen IL protocol (RAIL or TSIL), details are given later in this section, while explaining these protocols.

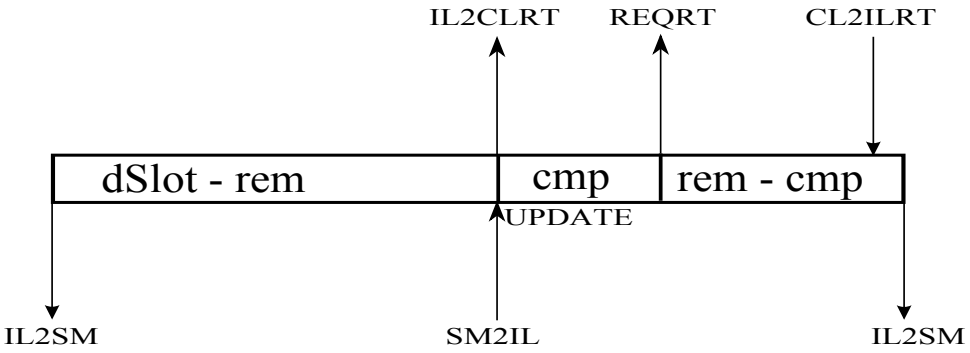

Figure 4.4: Actions in IL Protocol Family during one Time Slot

The six actions that are used to transmit messages are: IL2CLRT(m, t), CL2ILRT(b_1, b_2, m), IL2APNRT(q), AP2ILNRT(m), IL2SM(m), SM2IL(m). The former three are output actions, i.e., triggered by IL and used to pass the message to CL or SM. The later three, on the other hand, are input actions triggered by either CL or SM, and used to transfer message from them to IL. (Fig. 4.3)

Beside these six actions, used for message transmission, there are two other actions defined in the IL protocol family. The first one is the internal action UPDATE, that describes how to update the internal variables. The other one is the REQRT, used to request information from CL, which is replied by the CL2ILRT(b_1, b_2, m) action that carries information about the type (b_1) and ownership (b_2) of the next time slot, apart from the RT message (m). (Fig. 4.4)

Each slot in the IL protocol family's TDMA structure starts with the IL2SM(m) action. In this action, the device that owns the slot according to `myIL` variable sends the message m to SM. This message will be a RT message taken from `TxRT`, if the `RTIL` variable indicates that the time slot is a RT slot, or an nRT message received from `TxnRT` queue, otherwise. In either case, no message will be transmitted if the related buffer or queue is empty.

After $dSlot - rem$ space of time, which is enough for all devices to receive the transmitted message, SM2IL(m) action will take place, passing the received message m from SM to IL. Again, according to the variable `RTIL`, the received message will be put either `RxRT` or `RxnRT`. If the message is put to the `RxRT` buffer, right after, IL2CLRT(m, t) will occur, passing the message to the CL and cleaning the `RxRT` buffer.

The actions after this message transmission differ for protocols RAIL and TSIL, even their occurance stay same as shown in Fig. 4.4. Therefore, the rest of the time slot is explained in

two parts.

RAIL:

RAIL protocol assigns separate time slots statically for RT and nRT messages. This is done via the `RTSet` and `nRTSet` elements of the protocol specific structure `vIL`. `vIL.RTSet` is same for all devices in the system, setting those slots as RT slots, and the ownership of these slots are determined by the variable b_2, coming from CL, later with the CL2ILRT(b_1, b_2, m) action. The remaining slots are set as nRT slots and the devices in the system share them by uniquely including in their `vIL.nRTSet`.

This slot assignment repeats after `vIL.cyc` slots. In each UPDATE action, next slot's start time is set as

$$next = next + dSlot \tag{4.1}$$

and `vIL.cnt` increases with modulo `vIL.cyc` to count the time slots. If `vIL.cnt` is an element of `vIL.RTSet`, `reqIL` variable is set to trigger REQRT action to request information from upper layer, after *cmp* time, which is needed for actions SM2IL(m), IL2CL(m) and UPDATE. CL, as a response to REQRT, informs IL about if the next slot will be used for a RT message transmission (b_1) and if that RT message (m) is sent by this device (b_2), via the CL2ILRT(b_1, b_2, m) action. This action should occur at most $rem - cmp$ time after REQRT action, which is the dedicated duration for upper layer computations. Finally, the RT message will be transmitted with the next time slot, when now becomes equal to next.

If `vIL.cnt` is not an element of the `vIL.RTSet`, `reqIL` is not set and no REQRT action will be issued. Instead, in UPDATE action `vIL.cnt` is checked to see if it is in the `vIL.nRTSet`, which indicates that the next slot is assigned for this device's nRT message transmission. If so, when now becomes equal to next, the start of the next time slot, this time a message from `TxnRT` queue will be transmitted, if there is any.

TSIL:

Unlike the RAIL protocol, which assigns the type (RT or nRT) of the slots statically, TSIL protocol allows the CL to decide the type of each slot. Therefore, in each UPDATE action, `reqIL` variable is set to trigger the REQRT action to request information from CL about the

scheduling of the RT slots. CL, as a response to REQRT, issues the CL2ILRT(b_1, b_2, m) action to inform TSIL about the type (b_1) and the ownership (b_2) of the next slot. The timings of the actions REQRT and CL2ILRT(b_1, b_2, m) are still same as the ones in the RAIL. (Fig. 4.4)

The scheduling of the nRT slots, on the other hand, are done statically as the one in RAIL protocol. The slot counter vIL.cnt is increased with again modulo vIL.cyc, but only in the nRT slots. Therefore, vIL.cyc, this time indicates after how many nRT slots the nRT slot assignment repeats. These nRT slots are again shared uniquely between the devices in the system by including the slot numbers in their vIL.nRTSet. The remaining parts of the protocol are just same as the ones in RAIL.

4.3 Time-Slotted Implementation

As explained in the previous section, IL protocol family is responsible for message transmission in an accurate time-slotted operation. In the subsections of the present section, first the synchronization technique used for the implementation of TDMA (Time Division Multiple Access) structure of IL is explained briefly. Then, the best method to be used for accurate timings of the slots is chosen among the two alternatives: polling and interrupt. Finally, by describing how to start the TDMA structure simultaneously, this section is finalized.

4.3.1 Synchronization

Synchronization is the essential part of the TDMA structure. For implementing a correct time-slotted operation in a distributed system, all the nodes in the system should have their clocks synchronized as accurate as possible. In D^{3}RIP framework [6], the IEEE1588 [12], which is a standard defining a protocol enabling precise synchronization of clocks in measurement and control systems, is proposed as the synchronization technique.

The details of the IEEE1588 is out of the scope of this thesis, still a brief introduction is given to explain the modifications needed in the implementation of IL, for properly running an IEEE1588 application over a TDMA structure.

In IEEE1588 protocol, to calculate offset between the clocks there are four messages are defined (Fig. 4.5). Initially, the master node sends a *Sync* message, and while transmitting

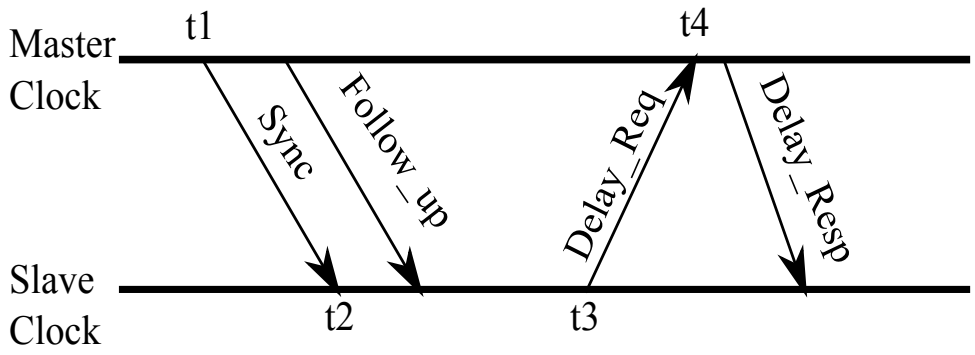

Figure 4.5: Messages of IEEE1588

this message takes a timestamp (t_1). When the slave node receives this message, it takes another timestamp (t_2). Then the master node send its timestamp (t_1) via the $Follow\_Up$ message. At this point the offset between the clocks can be calculated by

$$OffsetFromMaster = (t_2 - t_1) - MeanPathDelay \tag{4.2}$$

where $MeanPathDelay$ is the message transmission delay between slave and the master node and should be calculated by

$$MeanPathDelay = ((t_2 - t_1) + (t_4 - t_3))/2 \tag{4.3}$$

where t_3 is the timestamp taken by slave while transmitting the $Delay\_Req$ message and t_4 is the timestamp taken by the master when receiving that message. The timestamp (t_4) will be transmitted to the slave via the $Delay\_Resp$ message to calculate the offset and arrange its clock [13].

To run the IEEE1588 protocol as an nRT application, while implementing the IL protocol family, some modifications are necessary to the timestamping mechanism. In ideal case, this timestamping should be made as close to the shared medium as possible for precise synchronization. Therefore, the IEEE1588 applications take the timestamps just before sending to the network stack of the operating system. In the D^{3}RIP framework case, on the other hand, the IEEE1588 messages will not be transmitted immediately as they are sent out from the application, since the system runs over a TDMA structure. They will wait in the `TxnRT`

queue until a proper nRT slot arrives. This queuing delay is nondeterministic as it changes according to the nRT traffic and the time application sends the message. For example, if the synchronization application luckily sends the message just before the nRT slot that belongs to the device it runs. In this case, the message will be transmitted with the start of the slot and there will be no delay. If the it unfortunately sends the message just after the nRT slot that belongs to the device it runs, on the other hand, the message will wait for the next nRT slot that belongs to the device it runs, and this waiting duration depends on the scheduling. This jitter in the queuing delay results in fluctuations in the delay values of the IEEE1588 and results in malfunctioning in the synchronization.

To solve this problem, the queuing delay for the messages $Sync$ and $Delay\_Req$ are notified to the IEEE1588 application. The details of this solution are explained in subsection 4.4.1 while describing the $Fragmenter$. Note that, there is no modification need for other messages of IEEE1588, as the received messages are immediately sent to the upper layers.

4.3.2 Prioritizing nRT Packets

In implementations that are using nRT synchronization application, the nRT traffic density carries a great danger on the proper operation of the TDMA structure. If the nRT traffic is very high, the synchronization messages could be timed out. This situation will cause problems in the synchronization of the nodes and thus will result in the collapse of the whole TDMA structure.

To avoid this threat, synchronization packets should have priorities over other nRT packets. This prioritizing is accomplished by using two queues; one for nRT packets with high priority and one for the other ones.

When a nRT packet comes from Linux network stack to the first part of the transmit function of the Ethernet driver (see Fig.4.9), in the ”Send Packet to Fragmenter” step, if the packet is an IEEE1588 packet, it will be put to the high priority queue. Otherwise, it will be put to the low priority queue.

When a nRT slot for a device comes, in the popping a nRT packet from queue part (see Fig.4.12), IL Thread will first check the high priority queue. If it is empty, this time the IL Thread will check the low priority one.

4.3.3 Timing Accuracy

Apart from synchronization, which involves clocks of all the nodes in the system, each node should know its own time accurately. More accuracy in timings means less guard periods for time slots, which in turn increases throughput and decreases minimum achievable deadline for RT messages.

To decide the way to determine the time as exact as possible, firstly the two high resolution POSIX clock types of the Linux 2.6 kernel, i.e. "CLOCK_REALTIME" and "CLOCK_MONOTONIC" are examined. Then, three different methods to perform an action at the desired time are proposed and the best method is chosen.

4.3.3.1 High Precision Timer Types

The two virtual clocks inside the Linux 2.6 kernel are CLOCK_REALTIME and CLOCK_MONOTONIC [14]. The technical differences between these timers are given below in the Table 4.1.

CLOCK_REALTIME:

It is a virtual POSIX clock, which represents the machine's best-guess as to the current wall-clock, i.e., time-of-day time.

CLOCK_MONOTONIC:

It is a virtual POSIX clock, which represents the absolute elapsed wall-clock time since some arbitrary, fixed point in the past. It is not affected by changes in the system time-of-day clock.

Table 4.1: CLOCK_REALTIME vs. CLOCK_MONOTONIC

Timer Type	CLOCK_REALTIME	CLOCK_MONOTONIC
Can Jump	YES	NO
Can Slew	YES	YES
Can Generate Interrupt	YES	YES
Can Be Set To A Desired Value	YES	NO

As it can be observed from the Table 4.1 that CLOCK_MONOTONIC cannot be set to a desired value. In the previous section, it is indicated that to synchronize the clocks of the nodes

in the system, the offset between them is calculated; and according to this calculation, the clocks of the slave nodes should be modified. Therefore, the clock source for the implementation should be CLOCK_REALTIME.

4.3.3.2 Timing Methods

To perform an action at the desired time, two different methods are proposed: Polling the clock continuously until it reaches the desired time, and setting a timer interrupt to the desired time.

Polling:

Polling is another method to perform an action at a desired time. In this method, the time value of the clock is continuously read until the predefined desired time value is reached. This method achieves the best result in the means of accuracy; however, it keeps the processor busy until the desired time is reached. The IL implementation will have the greatest priority, since it provides services to the CL, which provides services to the RT applications. Therefore, keeping the processor all the time means that CL, RT applications and also nRT applications would not work; since in the implementation Linux with RT-preemption patch, which makes tasks with higher priority preempts those with lower priority, is chosen as the operating system as explained in Appendix B. As a result, this method is inappropriate to be used in the IL implementation.

Interrupt:

Last method to perform an action at a desired time is using a timer interrupt. In this method, a high resolution timer is connected to one of the clock sources mentioned in the subsection 4.3.3.1. When the pre-defined clock event, which can be an absolute time or a duration, occurs, the timer will raise an interrupt. In the interrupt handler of the timer, the thread waiting for performing an action can be informed either by a *semaphore* or a *wait_queue*. Then the thread will perform the action.

The performance of this timer interrupts is highly improved as the high-resolution timers get into the Linux kernel tree with the version 2.6.24. Besides, this method is not keeping the processor busy like polling method. Therefore, it is the best option for determining the time of an action to be performed.

4.3.4 Simultaneous Start of TDMA Operation

Apart from synchronization and timing accuracy, another issue for implementing a proper time-slotted structure is starting simultaneously, i.e., all nodes in the system should start the TDMA operation at the same time. Being a distributed system, in D^{3}RIP all the nodes keeps exactly the same scheduling. Thus, all the nodes know in which slot they will transmit. By implementing accurate synchronization and timing, the slot start times become close. Still if the nodes start the slotted behavior at different times, there will be shift between the scheduling of the nodes leading to collisions.

To overcome this problem, one of the nodes utilized as master node (similar to [15]) temporarily, to send a special SYNC message to identify the start of the communication cycle. This choice is done by the user, while setting up the industrial communication network. Therefore, using a temporary master for the start-up of the system does not violate the non-master-slave behavior of the real operation.

SYNC message indicates the start of the slotted structure; still the nodes will not start immediately after receiving this SYNC message, since they will receive it at different times, because of the transmission delay, and inaccuracies in the synchronization and timing. Instead, the receiver nodes will wait the start of the next ten seconds to start the TDMA structure. This will solve the problem unless the reception time is very close to the beginning of the next ten seconds. In that case, some nodes may receive the SYNC message in the next ten seconds, which causes ten seconds shift between the scheduling. To prevent this final problem, the master transmits the SYNC message in the middle of ten seconds.

4.4 Frame Segmentation

As D^{3}RIP framework depends on the TDMA structure with fixed time slots, the messages transmitted from IL to SM should have an upper boundary in size. To fulfill this requirement, in [6], it is proposed to choose the time slot duration long enough to transmit the longest RT message. Still nRT messages may cause trouble, since their size can be bigger than the longest RT message. Increasing slot size for transmitting the biggest Ethernet frame is undesirable, since it will increase the minimum delay for RT packets, which is the primary objective of the framework. Therefore, the nRT packets bigger than the longest RT message should be

fragmented into frames before transmission.

To fragment big packets into frames, a frame header structure is defined as shown in Fig. 4.6. The size of this structure is tried to keep as small as possible, since it will cause an overhead and decrease the throughput.

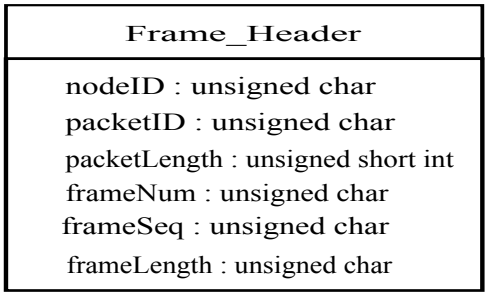

Figure 4.6: Frame_Header Structure

The elements of the Frame_Header structure and their use are;

nodeID:

- Holds the ID of the source node which transmits the packet.
- Chosen to be unsigned char (8 bit), which can support up to 256 nodes.

packetID:

- Holds the unique ID of the packet that is transmitted from the node with *nodeID*.
- Chosen to be unsigned char (8 bit) to identify up to 256 successive packets.

packetLength:

- Holds the total length of the unfragmented packet in Bytes.
- Chosen to be unsigned char (16 bit) to support maximum Ethernet frame, which is 1518 Byte according to [16].

$frameNum$:

- Holds the total frame number that the packet will be transmitted.
- Chosen to be unsigned char (8 bit) to support up to 256 frames that a packet with maximum length 1518 can be transmitted.

frameSeq:

- Holds the sequence number of the frame of a multi-framed packet.
- Chosen to be unsigned char (8 bit), since it can be maximum *frameNum*.

frameLength:

- Holds the data length in the frame in Bytes.
- Chosen to be unsigned char (8 bit) to support frames with up to 256 Byte data.

4.4.1 Fragmentation

To fragment the nRT packets, bigger than the maximum RT packet size, a *Fragmenter thread* is created. This thread waits for the nRT packets that will be transmitted via the Ethernet driver's transmit function, which is examined in the next section. When an nRT packet comes from the upper layer, this thread fragments it into frames and puts these frames to the `TxnRT` queue according to the flowchart given in Fig.4.7.

Fragmenter thread waits for an nRT packet from Ethernet driver's transmit function and when a packet comes, it checks the size of the packet to decide if it is needed to be fragmented. If the packet length is smaller or equal to the maximum RT packet size, *Fragmenter* puts it to the `TxnRT` queue only marking the packet as non-fragmented data coming from *Fragmenter*, not from the upper layer applications, to avoid it to come to *Fragmenter* again.

If the nRT packet's length is greater than the maximum RT packet length, i.e., the packet is needed to be fragmented, *Fragmenter* initially increases the *packetID*, which is used to distinguish the successive fragmented packets coming from one node. Then, *Fragmenter* calculates the number of frames that the packet will be fragmented into. At this point, *Fragmenter* starts preparing the frame header by assigning the elements that are common for all frames

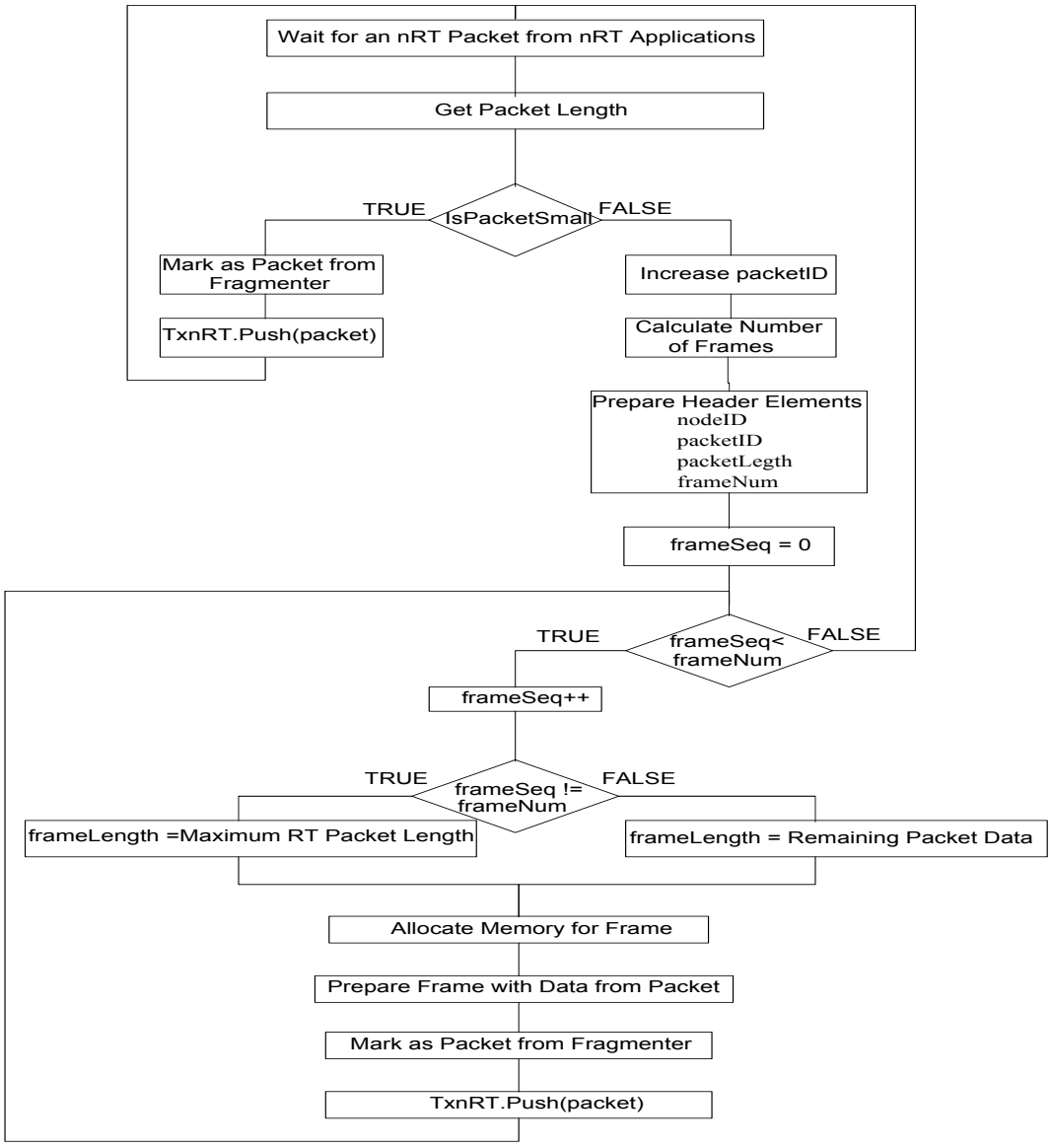

Figure 4.7: Flowchart of the Fragmenter Thread

of the packet, which are *nodeID*, *packetID*, *packetLength* and $frameNum$. Then, in a loop, $Fragmenter$ prepares the frames and sends them to the `TxnRT` queue. It, first prepares the rest of the header by increasing and setting the $frameSeq$, and determining the $frameLength$, which is

$$frameLength = maximumRTpaketsize - frameheadersize \tag{4.4}$$

for the initial frames and

$$frameLength = packetLength-(frameNum-1)*(maximumRTpaketsize-frameheadersize) \tag{4.5}$$

i.e., the remaining packet size, for the last frame. Besides, in this loop $Fragmenter$ allocates memory for the frame and copies the frame header and data from the packet's remaining part. In the final part of the loop, $Fragmenter$ puts the prepared frame to the `TxnRT` queue, again marking that it comes from $Fragmenter$. Finally, when the last frame is sent to the `TxnRT` queue, $Fragmenter$ breaks the loop and again waits for another packet from the Ethernet driver's transmit function.

4.4.2 Reassembly

To combine the fragmented packet frames coming from the Ethernet, before passing to the upper layers, a *Reassembler* thread is created. This thread waits for the fragmented nRT packet frames coming through the Ethernet driver's receive function, which is examined in the next section. As the *Reassembler* thread receives the frames, it combines these frames and transmit them to the upper layer via again the receive function of the Ethernet driver when a full packet is assembled, according to the flowchart given in Fig.4.8.

Reassembler thread waits for a frame of an nRT packet, which is fragmented in the $Fragmenter$ of the sender node. When an nRT packet frame is received by the Ethernet driver, the receiver function directly sends that frame to the *Reassembler* thread. After receiving a frame, *Reassembler* first resolves the frame header. Then, it looks at the *nodeID* element of the frame header to check if a receive session for a fragmented packet from that node contin-

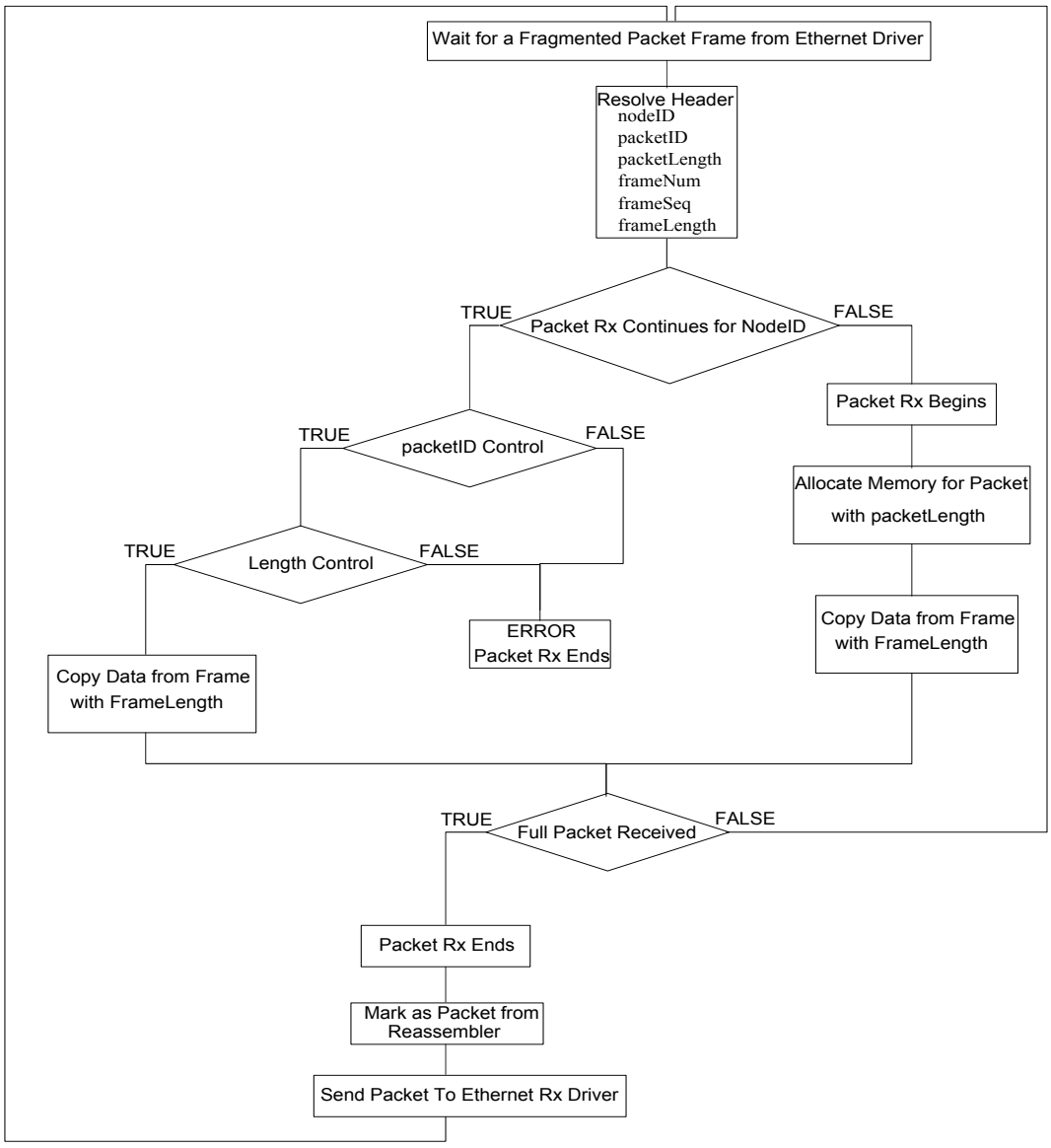

Figure 4.8: Flowchart of the Reassembler Thread

ues. If not, *Reassembler* allocates memory with the size of *packetLength* and sets that the receive session for the packet with *packetID* form node *nodeID* started. After copying data with size $frameLength$ from the frame's data part to the allocated memory for fragmented packet, *Reassembler* checks if the total received data length is equal to the *packetLength*. If so, *Reassembler* finalizes the receive session for a fragmented packet from that node and transmits the packet to the upper layer applications using the Ethernet driver's receive function. Just before sending the packet to the receive function, *Reassembler* marks that it comes from *Reassembler* to prevent the packet comes to itself again.

If the receive session for a fragmented packet from the node with *nodeID* continues, *Reassembler* first checks the *packetID* to verify that the received frame belongs to the packet that the receive session continues. Then *Reassembler* checks the $frameLength$ to see if the remaining part of the allocated memory is enough for the received frame's data. In both controls, if there is an error, *Reassembler* finalizes the receive session of the packet from the node with *nodeID* and releases the allocated memory for that packet.

If there is no error in the checks, *Reassembler* copies data with size $frameLength$ from the frame's data part to the allocated memory for fragmented packet. Finally, it checks if the total received data length is equal to the *packetLength*. If so, *Reassembler* finalizes the receive session for a fragmented packet from that node and transmits the packet to the upper layer applications using the Ethernet driver's receive function.

4.5 SM - IL Interface

For the communication between the SM and IL, transmit and receive functions of the Ethernet driver functions are used. Via these functions, the IL can transmit the messages directly to the Ethernet line and receive the packets or frames from Ethernet before they are transmitted to the network stack of the operating system. We present the implementation details for the transmit and receive functions in the remaining part of this Section.

Besides, these functions are used to get the nRT packets from the upper layer nRT applications and transmit the received nRT packets to them. To provide both of these capabilities; receive and transmit functions of the Ethernet driver are modified. Despite for different Ethernet card providers different driver codes exist, transmit and receive functions are similar. The reason

for this situation is that Linux operating system expectations for a network driver are uniform as explained in [17]. Only differences between different driver codes come from extra features provided, such as setting a maximum threshold for retransmission after a collision, provided by Intel Gigabit Ethernet Controllers.

The IL is designed to provide collision-free operation over shared medium Ethernet. However, incase of a fault and a subsequent failure of a device there can be collisions. By setting the maximum threshold for retransmission to zero, the retransmission mechanism of the Ethernet, which is a crucial threat for the correct operation of the TDMA structure, can be disabled. According to truncated binary exponential backoff algorithm [1] defined in the CSMA/CD standard [16], if a collision occurs, the sending node will retransmit its message after waiting for a random duration. This causes the node to transmit a message in an undetermined time, which causes the whole TDMA structure to be collapsed. In Intel's Gigabit Ethernet Controllers [18], there is a one Byte "Collision Threshold (CT)" entry in the "Transmit Control Register (TCTL)", which determines the number of attempts at re-transmission prior to giving up on the packet (not including the first transmission attempt). This entry gives an opportunity to the software developer to disable retransmission mechanism and its the main reason for the choice of Intel Gigabit Ethernet Controllers in the implementation of the D^3RIP framework. This is a simple preliminary reliability precaution. However, avoiding retransmission when an error occurs is not enough for the reliability of the framework, and further actions should be made to retrieve from the error condition, which is out of the scope of this thesis.

Thanks to the modular structure of Linux, these modifications do not need to be made directly on the kernel. Instead, the driver source codes, which can be obtained from both the kernel source codes and the driver providers web site, can be modified and build out of the kernel tree. After the start up, the Ethernet driver module can be removed and the kernel module output of the modified driver code can be inserted.

4.5.1 Transmit

The transmit function of the Ethernet driver is the final point of all the packets using the network stack of the Linux operating system. Besides, it is the function that transmits the Ethernet packets to the Ethernet line through the Ethernet controller. Therefore, with some modifications, this function is used as both the AP2ILNRT(m) function that receives all packets

coming from nRT applications and IL2SM(m) function that transmits RT and nRT packets to the Ethernet line.

Since this function is used for two different purposes, three Ethernet packet types are defined as Table 4.2.

Table 4.2: Ethernet Packet Types Defined for D^{3}RIP Framework

Packet ID	Packet Type
0x1100	IL_RT_PACKET
0x2200	IL_NRT_PACKET
0x3300	IL_SYNC_PACKET

The IL_RT_PACKET type is used for RT packets, IL_NRT_PACKET is used for fragmented nRT packet frames and IL_SYNC_PACKET is used for the SYNC message that is used to identify the start of the communication cycle as explained in subsection 4.3.4. As explained in [16], the value of the Length/Type field of the Ethernet header should be greater than 1536 decimal (0x0600 hexadecimal) in order to indicate Type interpretation; otherwise, it will indicate the number of data octets contained in the Ethernet packet. Therefore, all of these packet types are chosen to be greater than 0x600. Besides, while choosing these values for the packet types, paid attention to the already registered types at [19] and unique types are assigned.

As seen in Fig. 4.9, the transmit function of the Ethernet driver is modified in such a way that it supports the communication both before and after the *Fragmenter* thread is initialized. In the function, first it is checked that if the *Fragmenter* thread is initialized and if it is not, the modifications are bypassed and function acts as the original transmit function.

If the *Fragmenter* is initialized, then the Ethernet packet is checked if it is marked a non-fragmented packet coming from the *Fragmenter*. If it is not, the packet is checked to be an IEEE1588 packet coming from the network stack of Linux, this time. If it is an IEEE1588 *Sync* or *Delay_Req* packet, timestamp is taken for the arrival of the packet, in order to calculate the queuing delay further, and it is sent to the *Fragmenter* to wait for its time to be transmitted, not to be fragmented. If it is not an IEEE1588 *Sync* or *Delay_Req* packet, one final control is made to the Ethernet packet type to understand if it is an IL_RT_PACKET or

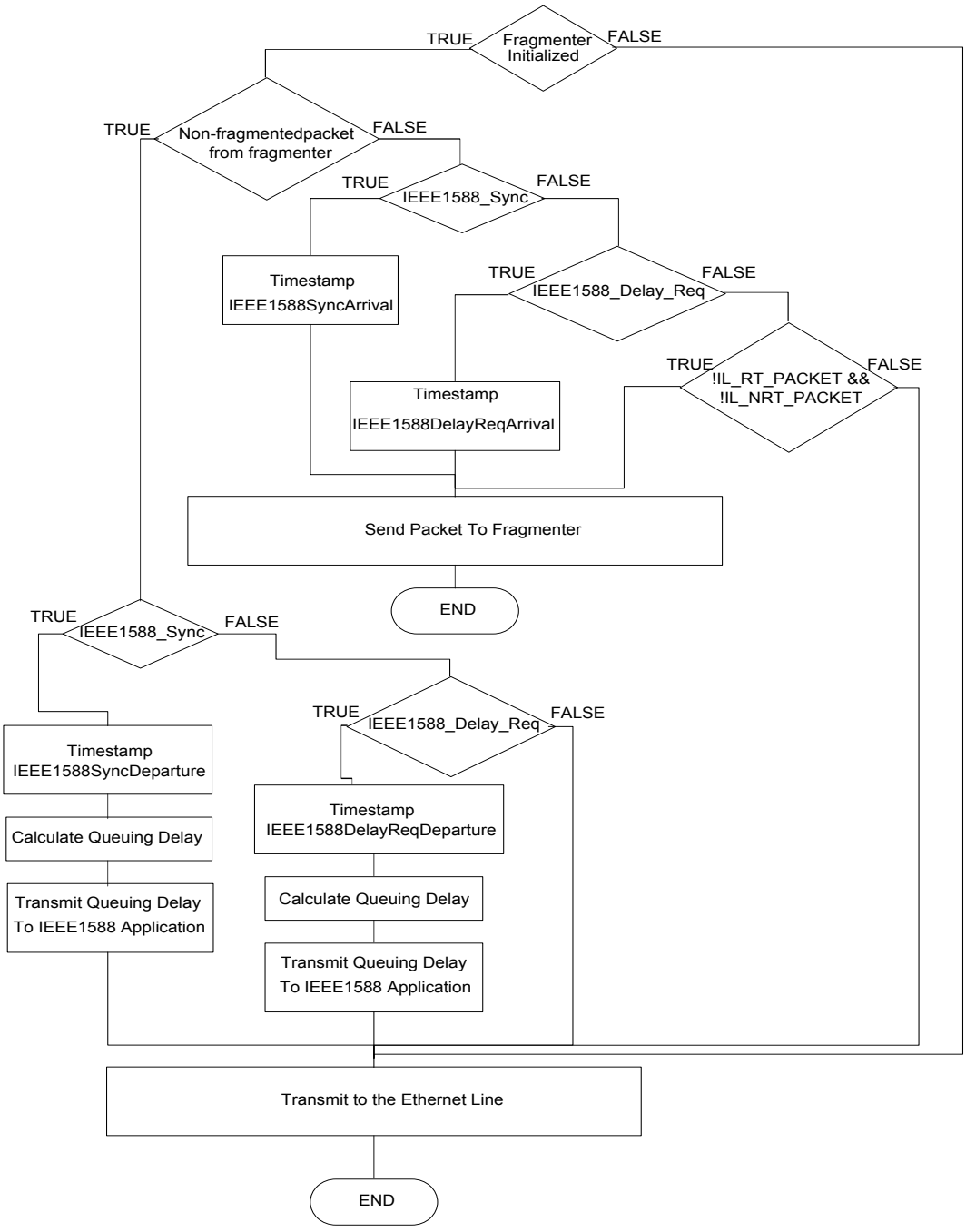

Figure 4.9: Flowchart of the Transmit Function

IL_NRT_PACKET. If so, it means that packet or frame is coming from the *Fragmenter* and it is transmitted to the Ethernet line via the unmodified part of the original Ethernet driver transmit function. (see subsection 4.3.1)

If the packet is marked as a non-fragmented packet coming from the *Fragmenter*, it should be a nRT packet smaller than the biggest RT packet. As it is explained in the next chapter, all the IEEE1588 packets are among this type of nRT packets. Even the packets are small enough that they do not need to be fragmented, they are initially sent to the Fragmenter thread to wait for a nRT slot to be sent. Therefore, at this point it is checked that if the packet is an IEEE1588 *Sync* or *Delay_Req* packet coming from the *Fragmenter*. If so, this time a timestamp is taken for the departure of the packet. The queuing delay that these packets subject to is calculated by the Eqn.4.6, and transmitted to the IEEE1588 synchronization application by using file operations of char devices [20]. The application will take this delay into account after receiving it [13]. Then, these packets are transmitted to the Ethernet line; again via the unmodified part of the original Ethernet driver transmit function. The non-fragmented packets coming from *Fragmenter*, other than IEEE1588 *Sync* and *Delay_Req* packets also transmitted via the unmodified part of the transmit function, without any delay calculation this time.

$$queuingDelay = departureTimeFromFragmenter - arrivalTimeToFragmenter \quad (4.6)$$

4.5.2 Receive

The receive function of the Ethernet driver is the first function that all the packets or frames coming from Ethernet controller arrives. Besides, it is the function that transmits these packets to the upper layer applications through the network stack of the operating system. Therefore, with some modifications, this function is used as both the SM2IL(m) function that receives all packets or frames coming from the Ethernet line and IL2APNRT(q) function that transmits nRT packets to the upper layer nRT applications. This is accomplished again by using the types defined in Table 4.2.

As seen in Fig. 4.10, the receive function of the Ethernet driver is modified in such a way that it supports all situations including IL protocol initialized or not, and *Reassembler* initialized

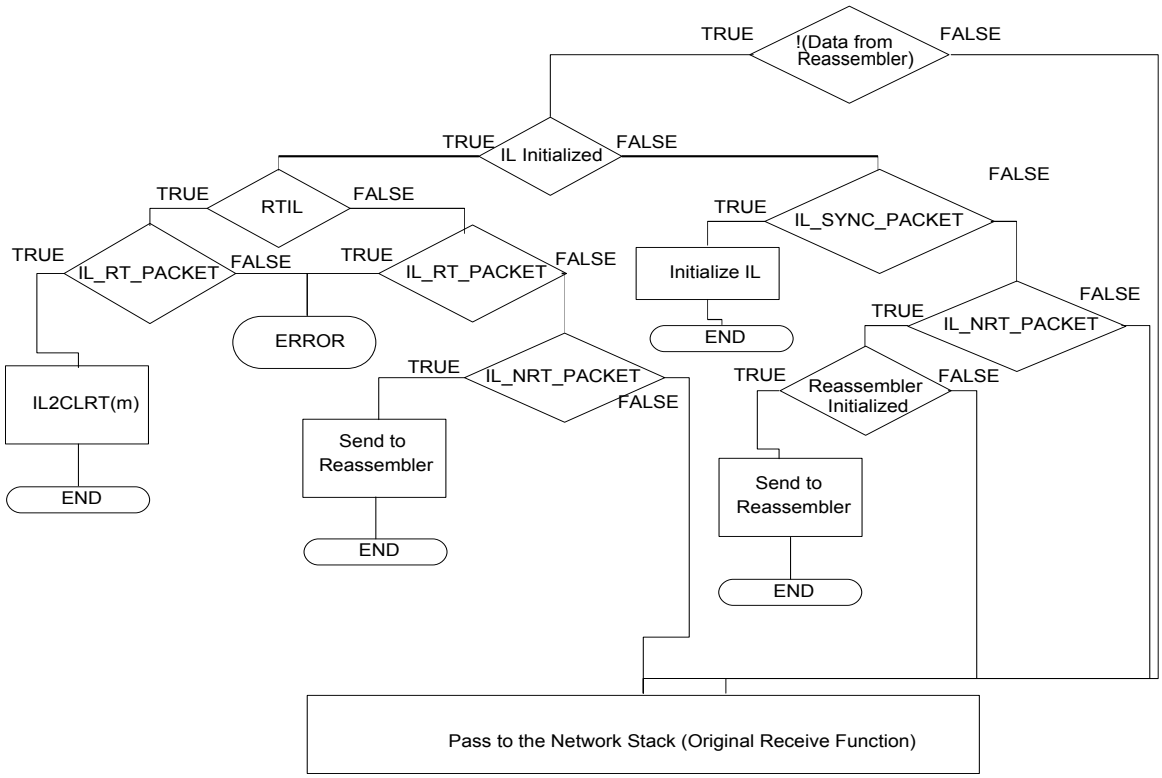

Figure 4.10: Flowchart of the Receive Function

or not. First it checks if the received packet or frame comes from the *Reassembler*. If so, the message will be transmitted to the upper layer applications through the network stack of the operating system, via the unmodified part of the original Ethernet driver receive function.

If the packet or frame is not coming from the *Reassembler*, the receive function checks if the IL protocol is initialized, which is done by temporary master (can be chosen as one of the devices in the system by the user) sending the special SYNC message to identify the start of the communication cycle and other nodes receive it, as explained earlier in subsection 4.3.4. If the IL protocol is not initialized yet, the first control is checking the Ethernet packet type to see if the received packet or frame is the SYNC message. If so, the receive function will inform the IL protocol to initialize by sending a *semaphore*. If the received packet or frame is not the IL_SYNC_PACKET message, the packet or frame is checked that if it is an IL_NRT_PACKET frame. Since *Fragmenter* and *Reassembler* threads are initialized earlier, fragmented nRT packet transmission can start before the IL protocol is initialized.

If the packet or frame is an IL_NRT_PACKET frame, it is controlled to see that if the *Reassembler* is initialized, since the transmitting node can initialize the *Fragmenter* before the receiver node initializes *Reassembler*. If the *Reassembler* is not initialized, the frame will be transmitted to the network stack of the operating system via the unmodified part of the receive function. As the Ethernet packet type is changed to IL_NRT_PACKET in the Fragmenter, the application that actually waits the packet could not receive it, which will resulted by the network stack's disposing the frame. If the *Reassembler* is initialized, the fragmented packet frame will be sent to the *Reassembler* to be combined and sent to the upper layer applications again using the receive function of the Ethernet driver.

If the packet or frame is neither IL_SYNC_PACKET nor IL_NRT_PACKET, and IL is not initialized, then it must be an non-fragmented nRT packet, i.e., nRT packet with a size smaller than maximum RT packet length, and will be immediately transmitted to the upper layer applications through the network stack of the operating system again using the unmodified part of the receive function.

If the IL protocol is initialized, then protocol specific actions are taken into account. If the `RTIL` variable is true, then the IL protocol expects a packet with type IL_RT_PACKET. If the Ethernet packet type mismatch, which means an error occurs, the packet is disposed. If the expected packet with type IL_RT_PACKET comes, it will be immediately forwarded to the

CL protocol family via the IL2CLRT(m, t) message.

If the RTIL variable is false, again the Ethernet packet type is checked. This time, since a nRT packet or frame is expected, if a packet with type IL_RT_PACKET comes, again it means an error occurred and the packet is discarded. If the Ethernet packet type is IL_NRT_PACKET, which means a fragmented RT packet frame came, the frame will be sent to the *Reassembler* to be combined and sent to the upper layer applications again using the receive function of the Ethernet driver. Otherwise, the received packet is a non-fragmented nRT packet and will be immediately transmitted to the upper layer nRT applications through the network stack of the operating system, via the rest of the receive function, which is the unmodified part of the original function.

4.6 IL - CL Interface

Apart from the interfaces between SM and nRT Applications, which are both realized by using the modified transmit and receive functions of the Ethernet driver; one other interface that should be implemented is between CL.

As it is needed to use the driver function in order to control all outgoing packets from the operating system, the IL protocol family is implemented in the kernel space. However, this necessity is not valid for the CL implementation. On the contrary, implementing it in the user space is more reasonable, since CL provides service to the RT applications and implementation in the user space is easier [21].

Table 4.3: Experimenter Setup Devices

Personal Computer (PC)	**Development Kit (DK)**
Hardware	*Hardware*
QuadCore Intel(R) Core(TM) i3 CPU 550@3.20GHz	Intel(R) Atom(TM) CPU Z530 @ 1.60GHz
3.6 GByte RAM	1 GByte RAM
Software	*Software*
Ubuntu (Release 10.10)	Ubuntu (Release 10.10)
Kernel Linux 2.6.33.7.2-rt30	Kernel Linux 2.6.33.7.2-rt30

We perform an experimental study to observe the timings of communication between IL and

CL when CL is implemented in kernel space and user space. The experiments are conducted on two devices with the properties given in Table 4.3.

The two basic requirements to be fulfilled are message passing, which means passing RT packets from IL to CL and vice versa, and signaling, which is IL's informing CL about its information request and CL's response.

4.6.1 Message Passing

Message passing is required for the RT packet transmission from CL to IL while transmitting, and from IL to CL when receiving. If CL is implemented as a user space application, to send messages from IL to CL *copy_to_user*() function should be used to copy data from kernel space to user space. To send messages from CL to IL, on the other hand, $copy\_from\_user()$ function should be used to copy data from user space to kernel space [22]. If CL is implemented in the kernel space, to transfer messages between IL and CL, the data is only needed to copied via the generic *memcpy*() function.

Therefore; to compare the speed of these three functions, a small size data with the length of 4 Bytes, a medium size data with the length of 150 Bytes and a large size data with the length of 10000 bytes (10KByte) are copied. Timestamps are taken just before and just after the copying process. Ten experiments are made by taking 10000 samples in each experiment and calculating the maximum delay. The average of these maximum values for two devices in Table 4.3 are given in Table 4.4.

Table 4.4: Data Copy Duration (in ns) with Different Techniques

	PC			DK		
	memcpy	copy_to_user	copy_from_user	memcpy	copy_to_user	copy_from_user
4 Bytes	881	18289	12745	5597	50604	82262
150 Bytes	1329	21700	17467	10127	46775	104582
10000 Bytes	15698	20907	49347	86389	108553	314409

As it can be seen from the results in Table 4.4, *memcpy*() has better performance than both *copy_to_user*() and $copy\_from\_user()$. This is an expected situation, since it can be seen form the source codes of the kernel in [23] that both of these functions include *memcpy*() function.

Furthermore, this functions access to the user space memory from kernel space. As stated in [20], the user pages being addressed might not be currently present in memory, and the virtual memory subsystem can put the process to sleep while the page is being transferred into place, which can happen, for example, when the page must be retrieved from swap space. As the memory being copied becomes greater, the time needed for copying becomes higher and the the ratio of the *copy_to_user*() and *copy_from_user*() duration to *memcpy*() duration becomes smaller. Still this difference should be considered in order to have a better performance.

4.6.2 Signaling

Signaling is required for CL and IL to inform each other about an action, like REQRT and CL2ILRT. If CL is implemented in the kernel space, IL and CL will notify each other via either a *semaphore* or a wait queue, which are the two methods used in kernel space for interprocess communications in Linux [24].

Otherwise, more complicated techniques should be used for communicating between kernel space and user space. The most widely-used one among these techniques is the file descriptors [25]. With *read* and *write* functions, CL can get data from and send data to IL, via the *copy_to_user*() and *copy_from_user*(), explained in the previous subsection. Still, using file descriptors directly are not enough for IL-CL communication, since there is no support for signalling to user space from kernel space. Therefore, either a separate signalling should be made or blocking read should be used.

As discussed in the [26], with the *send_sig_inf*() function a signal can be send to a specific process in the user space. By using this signal, IL can request upper layer information from CL or inform it that the RT message is ready to be read.

To compare the speed of signalling between kernel space - user space and kernel space - kernel space communications, five experiments using the methods above are conducted.

In the first two experiments, CL protocol is assumed to be implemented in the kernel space. In the first experiment, two threads, one corresponds IL and the other one corresponds CL, are created. One is waiting for a *semaphore*, which is send by the other thread. The sender thread takes a timestamp just before sending the *semaphore*, and the receiver takes just after receiving it. The difference between these timestamps gives the signalling delay

The second experiment is almost the same as the first one. The only difference is that instead of *semaphore*, the threads use *wait_queue* to communicate.

In the third, fourth, and fifth experiments, on the other hand, CL protocol is assumed to be implemented in the user space. The third experiment corresponds to the CL signalling to the IL via the *write* function of file descriptor. The thread, corresponding to the CL, is created in the user space and calls write function to send a message to IL. Since in the previous subsection the message copying delay is examined, in this experiment no copying is made. Only a *semaphore* is given in the *write* function to the IL thread implemented in the kernel space. In this experiment timestamps are taken just before calling the *write* function in the user space, and just after taking the *semaphore* in the kernel space.

The fourth experiment corresponds to the IL signalling to the CL. In this case, first the kernel thread, corresponding to IL sends a signal to the user space to inform about CL about an action. This signal triggers a user space function to give a *semaphore* to the user space thread, corresponding to the CL. Just after taking the *semaphore*, this thread calls *read* function of the file descriptor. Again since in the previous subsection the message copying delay is examined, in this experiment no copying is made. The timestamps are taken just before sending the signal and just after the *read* function returns.

The last experiment includes the usage of blocking read functions. In this case, in the user space, a thread is created to continuously read from the kernel space. The important point of this read is that it is blocking, which is accomplished by waiting a for a *wait_queue* in the definition of the read function in the kernel space. When the IL wants to signal CL, it simply release the *wait_queue*. Then, the read function will return and the thread calling the read function will call the user space implementation of the IL's action. Again since in the previous subsection the message copying delay is examined, in this experiment no copying is made. The timestamps are taken just before releasing the *wait_queue* and just after the read function returns.

Again ten experiments are made by taking 10000 samples in each experiment and calculating the maximum delay. The average of these maximum values for two devices are given in Table 4.5.

As it can be seen from the Table 4.5, all the signalling techniques experimented have similar

Table 4.5: Signalling Delay (in ns) with Different Techniques

	PC	DK
semaphore	18649	52492
wait_queue	27721	46458
write	28230	50972
read	27646	56512
blocking read	31295	50800

delay values. This shows that while communicating between user space and kernel space, there is not much delay apart from the paging delay mentioned in the previous subsection.

4.7 IL Thread Implementation

Considering the previous sections, we decide to implement IL as shown in Fig.4.11.

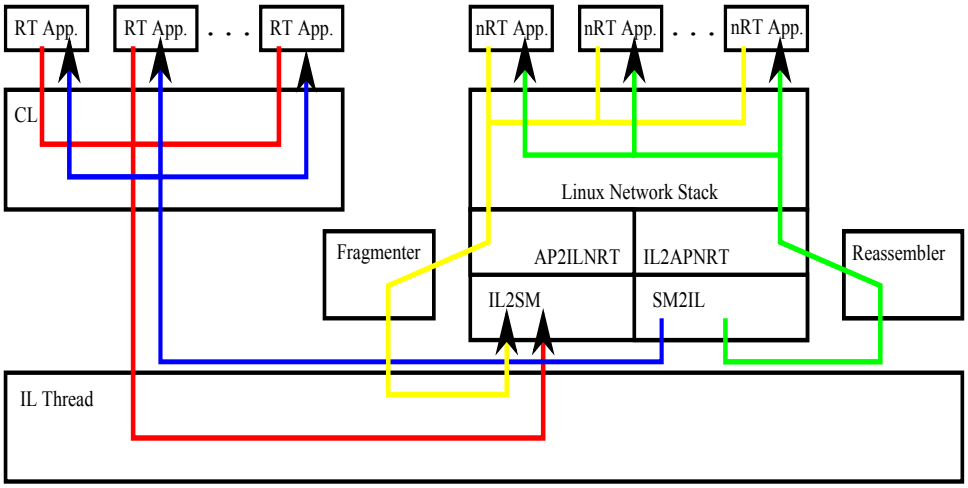

Figure 4.11: General Overview of IL Implementation

According to this implementation, the RT messages coming from the RT applications will follow the red line. First, they will be transmitted to the implemented CL protocol. Then, when CL decides that the next slot will be used for that device's RT transmission after a REQRT from the IL, CL will send an RT message to IL. When the next slot begins, IL will send it to the Ethernet controller via the IL2SM function, which is actually the second part of the Ethernet driver's modified transmit function.

On the receive side, the RT messages follow the blue line. When an RT message comes from the Ethernet line via the SM2IL(*m*) function, which is actually the second part of the Ethernet driver's modified receive function, only `RTIL` variable, which is controlled by IL and CL, is checked. If it is true, the RT message will be directly transferred to the CL. Otherwise, an error has occurred, and the system will stop.

The nRT messages coming from the nRT applications, on the other hand, will follow the yellow line. All messages to be transmitted to the Ethernet line will use the network stack of Linux, and the last point of this stack is the transmit function of the Ethernet driver. So, an nRT message will initially come to the AP2ILNRT function, which is actually the first part of the Ethernet driver's modified transmit function. Here, the nRT message will be transmitted to the *Fragmenter*, where it is put to the `TxnRT` queue of the IL Thread after being divided into frames if it is bigger than the maximum RT packet size. When IL decides that the next slot will be used for that device's nRT transmission, at the start of the next slot, a message from the `TxnRT` queue, if there is any, will be transmitted to the Ethernet controller via the IL2SM function, which is actually the second part of the Ethernet driver's modified transmit function.

On the receive side, the nRT messages follow the green line. When an nRT message comes from the Ethernet line via the SM2IL(*m*) function, which is actually the second part of the Ethernet driver's modified receive function, only `RTIL` variable, which is controlled by IL and CL, is checked. If it is false, the nRT message will be directly transferred to the *Reassembler*, where it is combined with the other frames if it a fragmented packet frame before transmitted to the upper layer nRT applications over the Linux network stack via the IL2APNRT function, which is actually the first part of the Ethernet driver's modified receive function. Otherwise, an error has occurred, and the system will stop.

Since the general working principle of IL is same for both protocols of the IL protocol family, RAIL and TSIL implementation realized in a similar manner. The only difference is in the IL Thread implementation, which controls the scheduling, as mentioned in section 4.2. Still, IL Thread implementation is not so different. As explained in the TIOA description in App.A.2, [6] defines protocol family as generic as possible. Therefore, the IL Thread is implemented in a generic way (Fig.4.12). The only difference between RAIL and TSIL is the implementation of the protocol specific functions as shown in Table A.1. Since these functions are only mak-

ing condition check and setting the corresponding variable, they do not have any difference in running time and implementation.

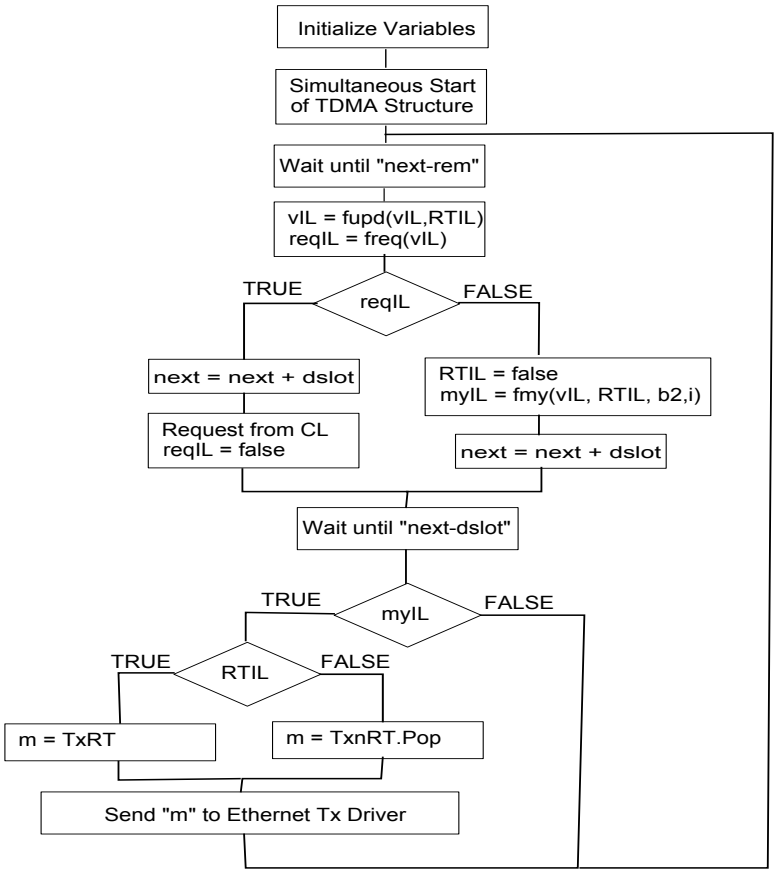

Figure 4.12: Flowchart of the IL Thread

As shown in Fig.4.12, the IL Thread, initialize the protocol related variables and waits for the simultaneous start of the TDMA structure at the start up, as explained in the subsection 4.3.4. When the time slots start synchronically, the IL Threads wait for the UPDATE time, which is "$next - rem$". When it is time to UPDATE, the protocol specific structure, `vIL` is updated via the protocol specific update function f_{upd}. Then it checks if there is a need for an upper layer information from CL by updating the variable `reqIL` by using another protocol specific function f_{req}. If the `reqIL` is false, which means that the next slot is not for RT transmission, IL Thread sets `RTIL` false, updates `myIL` via the protocol specific function f_{my} and updates the next slot start time *next*.

On the other hand, if the `reqIL` is true, which means that an upper layer information is needed,

only the next slot start time, *next*, is updated and the variables `RTIL` and `myIL` are not set in the scope of UPDATE. Instead, IL Thread sends a request to CL, sets `reqIL` false and waits until "$next - dSlot$". During this period, which is "$rem - cmp$", CL will responds with CL2ILRT function, which uses protocol specific functions f_{RT} and f_{my} to set the variables `RTIL` and `myIL`.

Then IL2SM part of the IL Thread takes place. Here, first `myIL` variable is checked to see if the next slot belongs to this device. If not, since reception is made out of IL Thread, it returns to the beginning to wait for the next UPDATE.

If `myIL` is true, IL Thread this time checks the `RTIL` variable to see if the slot belongs to an RT message or an nRT message. If it is false, IL Thread checks the `TxnRT` queue and sends one nRT packet or frame from this queue, if there is any. If it is true, on the other hand, IL Thread sends the message in the `TxRT` buffer, which is filled by the CL via the CL2ILRT function. In both cases, the transmission is made by the Ethernet driver's transmit function explained in subsection 4.5.1.

CHAPTER 5

PERFORMANCE EVALUATION and COMPARISON

5.1 Pre-Experiment Setup

Since the RAIL and TSIL protocols defined in the IL protocol family of the D^3RIP framework are implemented as explained in the previous chapter, the next step is conducting a long experiment to evaluate their performance. In the experiment, two PC's and two DK's, whose hardware and software properties are given in Table 4.3 are used.

These devices could be connected over either a switch, router of a hub. For real time communications, like ours, the time boundary for each message transfer should be explicitly known and the delays are intolerable. In switches and routers, there are decision processes. Switches decide the output port for the Ethernet packet according to the destination MAC addresses, and routers on the other hand have a more complex algorithm. Both switches and routers add some delay to the Ethernet packets because of these calculations. Also, in order to make these calculations, they contain some buffers, which make transmission delay nondeterministic. Therefore, they are inappropriate for real time communications.

Hubs on the other hand, are Layer 1 (Physical Layer) devices in the Open Systems Interconnection (OSI) model. They do not make any calculations on the packet. They simply receive the Ethernet frames, and broadcast these packets out from all other ports, by regenerating the electrical signal. So, there should be no packet delay resulted by the calculations. According to [27], the delay jitter in hubs without congestion is about 100ns at most. Since this value is too small, hubs will not be a burden for the deadline calculations of the messages. Therefore, hubs are the most appropriate network connection equipments. As a result, the two PC's and two DK's are connected over a 10Mbps Hub.

5.2 Assumptions

Before starting the experiment, some calculations should be made to determine the slot duration to be used in the final experiment. First thing to be calculated is the maximum message size to be sent in one slot. Actually, this value should be equal to the maximum RT packet that can be transmitted. Since the CL protocol family is out of the scope of this thesis, the maximum RT packet size could not be known. Therefore, another approach is made, and slot size is chosen to be big enough for all synchronization messages, which are the only nRT packets that are prerequisite for the operation of the TDMA structure implemented by the IL protocol family. From the IEEE1588 synchronization message sizes given in Table 5.1, it can be seen that the biggest message is the *ANNOUNCE* message, which is used to announce that a new node is joining the synchronization network, with a length of 106 Bytes. Therefore, by rounding up, a slot size with a capacity to transmit 150 Bytes is assumed. Thus, by subtracting the 14 Bytes MAC header overhead (according to [16]), the maximum RT message size is 136 Bytes.

Table 5.1: Lengths of the IEEE1588 Synchronization Messages Used in the Experiments

SYNCH	86 Bytes
FOLLOW_UP	86 Bytes
DELAY_REQ	86 Bytes
DELAY_RESP	96 Bytes
ANNOUNCE	106 Bytes

The second parameter to be calculated is the maximum value of *rem*, the part of the time slot that is not used for message transmission. During this period, IL will update its internal variables (UPDATE), and request information from CL (REQRT) and wait for the response of CL (CL2ILRT(b_1, b_2, m)). Assuming that CL is implemented in application layer, and using *blockingread* and *write* techniques to communicate, the *rem* duration can be approximated.

For REQRT and IL2CLRT(m, t), *write* technique is assumed to be used. To distinguish them, the first Byte of the written data is used. Since in REQRT no message transmission is made, the amount of the memory copied is 1 Byte. The IL2CLRT(m, t), on the other hand, copies an RT packet of 136 Bytes. Thus the amount of the memory copied is 137 Bytes.

Considering the maximum values (using 4 Bytes instead of 1 Byte and 150 Bytes instead of 136 Bytes) in Table 4.4 and Table 4.5 in the Sec4.6, the time for REQRT and CL2ILRT(b_1, b_2, m) can be approximated as:

REQRT time for PC:

$$28230 + 18289 = 46519ns$$

REQRT time for DK:

$$50972 + 50604 = 101576ns$$

CL2ILRT(b_1, b_2, m) time for PC:

$$31295 + 17467 = 48762ns$$

CL2ILRT(b_1, b_2, m) time for DK:

$$50800 + 104582 = 155382ns$$

As the UPDATE is only composed of basic C operations, it is negligible. Therefore, the maximum value of *rem* should be greater than 95281 for PC's and 256958ns for DK's.

The last parameter to be calculated is the maximum value of $dSlot - rem$, which is the time needed for the message transmission.

Since in the experiments 10Mbps hub is used, the transmission delay should be at least

$$150Bytes \times (8bits/1Byte) \div 10000000bits/sec = 0.12msec$$

Apart from this transmission time, there is also a negligible propagation delay factor caused by the used cable lengths. Besides, the mismatches in the clocks caused by the IEEE1588 synchronization tolerance, which is about 10-100us in software implementations according to [28], should be considered. Final source of delay is the memory copying. The message is copied from IL to SM via IL2SM(m), which uses *memcpy*() and a *semaphore*, at the transmitting node and is copied from SM to IL on the receiver node via SM2IL(m), which uses *memcpy*() and a *semaphore*, and then immediately it is transferred to the CL via IL2CLRT(m, t), which uses *write* as mentioned earlier.

Considering the maximum values in Table 4.4 and Table 4.5 in the Sec4.6, the time for IL2SM(m), SM2IL(m) and IL2CLRT(m, t) can be approximated as:

IL2SM(m) time for PC:

$$18649 + 1329 = 19978ns$$

IL2SM(m) time for DK:

$$52492 + 10127 = 62619ns$$

SM2IL(m) time for PC:

$$18649 + 1329 = 19978ns$$

SM2IL(m) time for DK:

$$52492 + 10127 = 62619ns$$

IL2CLRT(m, t) time for PC:

$$31295 + 21700 = 52995ns$$

IL2CLRT(m, t) time for DK:

$$50800 + 46775 = 97575ns$$

The total measured time needed for message transmission is 312951 ns for PC's and 442813 ns for DK's

Apart from these measured delays, also there are bus delays that are needed for the message to be put on the Ethernet line after the driver's transmit function and that are needed for the message to be received from the Ethernet line before the driver's receive function.

Since the experiments in Sec4.6 are short term experiments, the values should be chosen much higher for the long-term experiment, which is conducted to obtain much precise values. Since in the experiment both PC's and DK's are used, the maximum value of the *rem* and $dSlot - rem$ should be chosen according to the DK, which has worse performance.

As a result, for the experiment, the *rem* value is assumed to be 2 ms, which is about 8 times greater than the measured maximum value. The $dSlot - rem$ value, on the other hand, is assumed to be 8 ms, which is far greater, since there are delay sources that could not be measured.

As an important point, it should be kept in mind that that the parameters above are calculated for a specific configuration with PC's connected over a 10Mbps hub. Besides it is assumed that the maximum RT message size is smaller than 136 Bytes. These parameters would be different for different configurations and different maximum RT message sizes. Therefore, for different configurations, the performance will be different.

5.3 Pre-Experiment

The purpose of this experiment is determining the slot duration of the TDMA structure generated in the IL implementation of the D^{3}RIP framework as precise as possible. Thus, the performance metrics defined in Sec.3.2 can be calculated in the next section.

In the experiment, in the light of the calculations in the previous section, *rem* and $dSlot - rem$ values are assumed to be smaller than the values 2 ms and 8 ms respectively. To see the performance of each node in the experiment, a fair scheduling is assumed with one RT and one nRT slot for each node (see Fig.5.1). The CL part is simulated. Therefore, RAIL protocol is chosen to be used in the experiment in order to keep the simulation part as simple as possible. Because, in RAIL, separate time slots are assigned statically for RT and nRT messages, whereas in TSIL, CL decides both the type and ownership of each slot. Furthermore, the results will not differ for RAIL and TSIL, since the only difference of them is in the functions that consist of only the basic C operations, which are negligible in terms of time needed, when compared with the inter-protocol communications and transmission duration.

nRT Slot for PC-1	RT Slot for PC-1	nRT Slot for DK-1	RT Slot for DK-1	nRT Slot for PC-2	RT Slot for PC-2	nRT Slot for DK-2	RT Slot for DK-2

Figure 5.1: The Fair Scheduling of the Experiment

The nRT slots are used for the synchronization messages, whereas the RT slots will be used dummy RT messages that are time-stamped in order to measure the actual maximum values of *rem* and $dSlot - rem$ durations.

The experiment is run for 4 hours. Since the slot time is 10 ms and each node has one RT

slot out of eight slots, each node have transmitted 180000 RT messages. Total of 720000 RT messages are time-stamped. As a result of these experiment, the maximum value of time needed for the message transmission, i.e., $dSlot-rem$ can be found in Table 5.2. Furthermore; the maximum value of *rem* time can be found at Table 5.3.

Table 5.2: Experimentally Calculated $dSlot - rem$ Duration(in ns)

PC-PC	PC-DK	DK-PC	DK-DK
1323805	3892970	1239851	3889438

Table 5.3: Experimentally Calculated *rem* Duration(in ns)

PC	DK
244361	850423

As it can be seen from Table 5.2 and Table 5.3, worse hardware had worse performance. In a network consist of only PC's, 3 ms time slot with 0.5 ms *rem* and 2.5 ms $dSlot - rem$ could work even if the maximum values are doubled. Adding only a DK, on the other hand, could even drop the performance drastically by increasing the time slot duration to 10 ms with 2 ms *rem* and 8 ms $dSlot - rem$, if the guard period with same ratio is used.

Therefore, in the next section, the calculations are made considering a network with good hardware with 3 ms slot duration in order to obtain better results.

5.4 Calculations

According to the results of the experiment conducted in the previous section, the performance metrics mentioned in Sec.3.2 are calculated separately.

5.4.1 Throughput

The minimum slot duration obtained in the previous section is 3 ms on a network that consists of PC's connected over a 10Mbps hub. The data transmitted in each slot is assumed to be 150 Bytes. Therefore, the throughput can be calculated as:

$$(150Bytes \times 8bits/Byte) \div 0.003 = 0.4Mbps$$

5.4.2 Maximum RT Throughput

Even if there is no nRT communication load, some slots should be allocated for nRT transmission of the synchronization packets. The default periods of the IEEE1588 synchronization messages are given in Table 5.4.

Table 5.4: Default Periods of the IEEE1588 Synchronization Messages

SYNCH	1 second
FOLLOW_UP	1 second
DELAY_REQ	8 seconds
DELAY_RESP	8 seconds

According to these values, in each 8 seconds, the IEEE1588 master node transmits 8 $SYNCH$, 8 $FOLLOW\_UP$ and 1 $DELAY\_RESP$ messages, while the IEEE1588 slave nodes transmit 1 $DELAY\_REQ$ message. In a "N" node system, total nRT slots should be allocated is:

$$8 + 8 + 1 + (N - 1) = 16 + N$$

Assuming a four node industrial communication network, at least 20 slots in 8 seconds should be assigned to nRT packets for the correct operation of the protocol. To work with integers; in 24 seconds (8000 slots with 3ms duration), 60 slots should be assigned to nRT messages. Therefore, the maximum RT throughput is:

$$((8000 - 60) \div 8000) \times (0.4Mbps) = 0.397Mbps$$

5.4.3 Efficiency

The time that is not used for the message transmission and the time needed for the synchronization message transmission drop the efficiency of the IL protocol family of the D^{3}RIP

framework. As calculated in the previous subsection, the maximum RT throughput can be achieved in a four node industrial communication connected over a 10Mbps hub is 0.397 Mbps. Therefore, its efficiency is:

$$0.397Mbps \div 10Mbps = 3.97\%$$

This poor efficiency is for the specific configuration used in the pre-experiment. With some modifications on the configuration, like using bigger RT messages and better hardware devices, this value could be increased. Actually, the major reason behind this inefficiency is that using a non-Real-time operating system. Even the RT-preemption patch is used, the overall behavior of the Linux is nondeterministic. Because of this nondeterminism, the results of the pre-experiment have huge jitter. Thus the biggest values of the results are used and even that values are doubled for calculating the slot size in order to prevent possible errors.

5.4.4 Minimum RT Message Deadline

This metric is highly dependent on scheduling. For TSIL protocol, for example, all scheduling is done in the CL. Therefore, when an urgent RT message is to be transmitted, CL will allocate the next slot for that message's transmission. In the worst case, the urgent message will come just after CL replied to IL's request. At that time, since CL has already assigned the next slot, the urgent message will wait *rem* duration for the start of the next slot, $dSlot$ time for the transmission of the already assigned message and $dSlot - rem$ duration for the transmission of itself (see Eqn.5.1).

$$TSILs\_Minimum\_RT\_Message\_Deadline = 2 \times dSlot \tag{5.1}$$

As a result, in the configuration described above, TSIL protocol can provide service to RT Messages whose minimum deadline is at least:

$$2 \times 3ms = 6ms$$

RAIL protocol, on the other hand, assigns separate time slots statically for RT and nRT mes-

sages. This static allocation of nRT slots makes urgent RT messages to wait during nRT slots. In the worst case, the urgent message will come just after CL replied to IL's request, and maximum number of consecutive nRT slots are scheduled after the already assigned RT slot. At that time, since CL has already assigned the next slot, the urgent message will wait *rem* duration for the start of the next RT slot, $dSlot$ time for the transmission of the already assigned RT message, $maximum\_number\_of\_consecutive\_nRT\_slots \times dSlot$ duration for the transmission of the nRT messages and $dSlot - rem$ duration for the transmission of itself (see Eqn.5.2).

$$RAILs\_Minimum\_RT\_Message\_Deadline = (maximum\_number\_of\_consecutive\_nRT\_slots+2)\times dSlot \tag{5.2}$$

For example, if the scheduling of the industrial communication network is as in the Fig.5.1, in the worst case, RAIL protocol can provide service to RT Messages whose minimum deadline is at least:

$$(1 + 2) \times 3ms = 9ms$$

TSIL's better performance originate from its ability to adapt the RT and nRT communication needs of the network by assigning the slots to them dynamically. RAIL, on the other hand, statically allocates time slots for RT and nRT communication, even if there is no message of that kind to be sent, which drops the performance.

nRT Slot for PC-1	nRT Slot for DK-1	nRT Slot for PC-2	nRT Slot for DK-2	RT Slot for PC-1	RT Slot for DK-1	RT Slot for PC-2	RT Slot for DK-2

Figure 5.2: The Alternative Scheduling of the Experiment

Still in RAIL, with some precautions, the performance of IL, in terms of minimum RT message deadline, could be prevented from decreasing more. First, number of slots assigned to nRT messages should be kept in minimum. Second, these nRT slots should not be assigned consecutively. For example, if the scheduling is assumed to have same number of RT and nRT

slots, but in a different sorting as shown in Fig.5.2, in the worst case an urgent RT message will have to wait for four nRT slots. This results in a performance drop by increasing the minimum RT message deadline to:

$$(4 + 2) \times 3ms = 18ms$$

Apart from TSIL's performance advantage against RAIL, a point to be considered in TSIL is that it is highly dependent on the CL protocol and pays minimum attention to the nRT communication. This situation could result in a failure in the implementations that use nRT applications for the synchronization. If the RT communication becomes dense for some time, even the RT messages with high deadline would be transmitted instead of nRT messages. Thus, the synchronization protocol will starve and malfunction, putting the whole TDMA structure in danger.

5.5 Final Experiment

After determining proper values for the parameters in the previous section, a final experiment is conducted to see the performance of the D^3RIP framework in a simple industrial system. As an industrial system example, the system with two controller and a plant, which is described in [13], is used.

In this system, the plant is simulated on a DK and two controllers are implemented on separate PC's. The communication between plant-controller_A and plant-controller_B are realized via serial connection. The communication between controller_A-controller_B, on the other hand, is realized through 10Mbps hub by using D^3RIP framework. RAIL is chosen as the IL protocol, whereas URT is chosen as the CL protocol. The maximum RT message size is selected to be 136 Bytes in order to use the parameters calculated in the previous section. Therefore, slot size with 3 ms *dSlot* and 0.5 ms *rem* duration will be appropriate. The slots in the experiment are scheduled as shown in Fig.5.3.

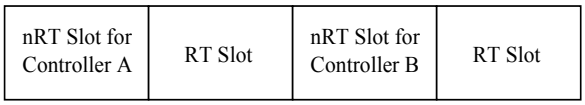

Figure 5.3: The Scheduling of the Final Experiment

In the example industrial system used in the final experiment, there are only three RT messages using the D^{3}RIP framework. At random times, the plant triggers the messaging shown in Fig.5.4. In each messaging Controller_B sends a message (?*mue*) to the Controller_A. Controller_A as a response to this message, sends another message (!*mue*) to the Controller_B. Finally, Contoller_B responses this message with a different message (*mue*), and the messaging ends until the plant trigs another one. The details of these messages can be found in [13].

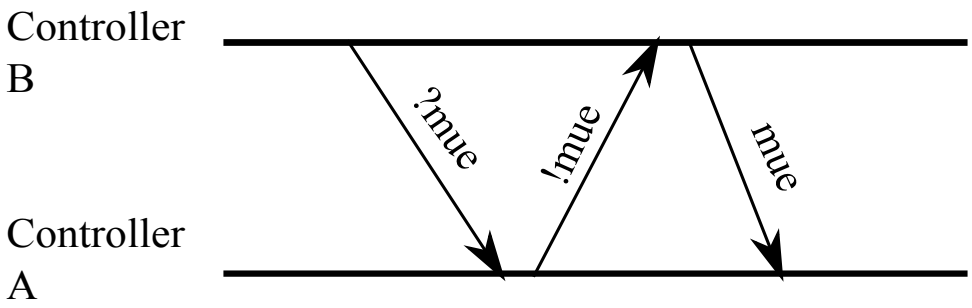

Figure 5.4: The RT Messages of the Final Experiment

This experiment is run for 24 hours under full nRT packet load. From each PC, a pinging session to the other PC with 20 ms interval and 150 Bytes total Ethernet packet size is started. Since each PC tries to reply the ping request coming from the other one, total of 4 nRT packet is tried to be sent in 20 ms. According to the scheduling shown in Fig.5.4 and 3 ms slot duration, 4 nRT slots are allocated in 24 ms. Therefore, the ping packets overload the allocated nRT slots. Even under that much nRT load, the D^{3}RIP framework worked free of error.

In this experiment, the latency of a RT message caused by the D^{3}RIP framework is measured. To measure this latency, two timestamps are taken: One is just after the application sends the message to the URT, and the other one is just after the RAIL sends the message to the driver.

The maximum and minimum values of this duration is given in Table 5.5.

Table 5.5: RT message latency caused by D^{3}RIP framework (in ns)

	?mue	!mue	mue
Maximum	6572990	5265931	5399695
Minimum	486797	4055472	4820411

In the worst case, the RT message will come just after URT replied to RAIL's request. At that time, since URT has already assigned the next slot, the RT message will wait *rem* duration for the start of the next RT slot, *dS lot* time for the transmission of the already assigned RT message, and *dS lot* duration for the transmission of the nRT messages before the RAIL sends the message to the driver. Therefore, the maximum latency value is expected to be about

$$0.5ms + 3ms + 3ms = 6.5ms$$

In the best case, on the other hand, the RT message will come just before URT replies to RAIL's request. At that time, URT will immediately assign the next slot for the transmission of this RT message. Therefore, it will wait only *rem* duration before the RAIL sends the message to the driver. Therefore, the minimum latency value is expected to be about

$$0.5ms$$

Since the ?*mue* message is transmitted randomly, it catches both the worst and best cases. However, the maximum value is measured to be a little greater than the theoretical maximum value. This slight error is caused by the timestamping inaccuracies. Besides, the minimum value is measured to be a little smaller than the theoretical minimum value. The reason of this slight error, on the other hand, is the timer interrupt inaccuracy that is used to determine the time of an action to be performed, as explained in the subsection 4.3.3.

The !*mue* and *mue* messages, on the other hand, could not catch the worst or best cases, since they are not transmitted randomly. On the contrary, they are transmitted by the application just after it received the ?*mue* and !*mue* messages. Therefore, they are transmitted at some time between the start of a RT slot and *dS lot* − *rem* time after the start of a RT slot. Thus,

these response messages will wait a time between *rem* and *dS lot* until the RT slot ends. Then they will wait for *dS lot* time for the transmission of the nRT messages. So the latency values for the !*mue* and *mue* messages is expected to be between

$$0.5ms + 3ms = 3.5ms$$

and

$$3ms + 3ms = 6ms$$

Besides, from the latency values of the !*mue* and *mue* messages, the time needed for the transmission of these RT messages can be calculated. Since the maximum latency value is about 5.4 ms (for *mue*), the RT message before that message (!*mue*) should be received

$$5.4ms - 3ms = 2.4ms$$

before the end of the slot. Therefore, the minimum transmission time of a RT message (!*mue*) is

$$3ms - 2.4ms = 0.6ms$$

Since the minimum latency value is about 4 ms (for !*mue*), the RT message before that message (?*mue*) should be received

$$4ms - 3ms = 1ms$$

before the end of the slot. Therefore, the minimum transmission time of a RT message (?*mue*) is

$$3ms - 1ms = 2ms$$

CHAPTER 6

CONCLUSION and FUTURE WORK

6.1 Conclusion

The improvements in the technology make a demand in the industrial control area for faster communication techniques. Traditional industrial communication techniques, i.e., proprietary fieldbuses are developed in such a strict way that they could not adapt to the needs of the industry. Hence, the recent studies on this area focus on the use of Ethernet as an industrial communication technique. The primary reason of this choice is that Ethernet has proved its success in the home and office for the last years, being the most widely used networking technique. This widely usage results in an increase in the researches on Ethernet. Thus its cost is decreasing and its speed is increasing further and further.

Still, the truncated binary exponential backoff algorithm, which is used in Ethernet after the occurrence of a collision, causes randomness. Thus, it cannot be used in industrial communication networks as itself.

In the literature, various studies address this problem and provide different types of solutions. Protocols such as Ethercat, SERCOS III, and ProfiNet, suggests hardware modifications on the MAC layer, which causes high cost and incompatibility. Protocols such as MODBUS RTPS and PROFINET SRT, aims to decrease the collision probability and reaction times. Still they could not provide real-time guarantees but only performance improvements. Different from that, protocols such as Ethernet/Ip (EIP) use switches to prevent collisions. This type of solutions, on the other hand, struggle with different problems like queuing delays and message loss because of the limited queue size. The most common solution is the avoidance of collisions by a specialized protocol on top of the standard Ethernet. The techniques in

this type of solution use master-slave (EPL), token passing (TTCNet), or static TDMA (EPA) in the additional layer. The techniques in this solution suffer from different disadvantages. Master-slave solutions have single point of failure, undistributed structure and low efficiency. Token passing solutions lack dependability, especially in the case of losing the token. Static TDMA solutions have low efficiency due to the guard periods and error recovery precautions.

The reason behind the inefficiencies is that the techniques in the literature make worst-case assumptions, such as all messages sent at the same time, and capacity allocation is done accordingly. However, in industrial control systems, the communication requirements of the system components are known at any time, since automation control applications work deterministically. Consequently, the D^3RIP framework is developed considering these characteristics of the industrial communication networks

In this thesis, the implementation of IL protocol family of the D^3RIP framework is realized on two different hardware platforms. The challenges encountered on the implementation, such as synchronization over TDMA structure, prioritizing synchronization packets, minimizing the guard periods in the time slots, starting the TDMA structure simultaneously, fragmentation and reassembly of big nRT packets, interfaces with SM and CL, are overcome successfully as described below.

- To realize an accurate synchronization over TDMA structure, the queuing delay that synchronization packets expose to is calculated and the IEEE1588 application is informed about this delay.

- To prioritize synchronization packets, a structure with two queues, one for high priority packets and one for the low priority packets, is implemented for nRT packets.

- To minimize the guard periods in the time slots, the two high resolution POSIX clock types of the Linux 2.6 kernel, i.e. ”CLOCK_REALTIME” and ”CLOCK_MONOTONIC” are examined, and three different methods to perform an action at the desired time are proposed and experimented.

- To start the TDMA structure simultaneously, one of the nodes utilized as master node temporarily to send a special SYNC message to identify the start of the communication cycle.

- For providing service to the nRT packets that are greater than the slot size, frame segmentation structure is constructed.

- In order to transmit and receive with minimum delay, which will increase the performance of the TDMA structure, interface between SM and IL is constructed by modifying the transmit and receive functions of the Ethernet driver.

- To realize the interface between CL and IL, three different techniques for message passing and five different techniques for signalling are inspected and experimented.

According to the experiment conducted in Sec.5.3, even the implementation is realized in a generic way, used hardware platform directly affects its performance. In DK, time slot duration that can be obtained is about 10 ms; whereas on PC, this value could drop to 3 ms.

Besides, it is seen that the reason of the poor performance of 3.97% is mainly due to the used operating system. For example, even in the implementation on good hardware devices, total message transmission duration is about 10 times more than the actual transmission time of the message on the Ethernet line. Because, even if RT-preemption patch is used on Linux, Appendix B shows that it could not obtain hard real-time performance.

6.2 Future Work

6.2.1 Implementation on Different Hardware or Operating System

As explained in the conclusion section 6.1, the poor performance obtained in the experiment in section 5.3 is mainly resulted by the used hardware and the operating system. To see the actual performance of the D^3RIP framework, it could be implemented on a real-time operating system, such as vxWorks [29].

Furthermore, implementation of D^3RIP framework could be realized on a Field Programmable Gate Array (FPGA) based hardware instead of a processor based hardware. This implementation could be more difficult, but it will give more precise performance measures by keeping the non-protocol related delays at minimum.

6.2.2 Automatic Slot Duration Determination

The slot duration to be used in the D^{3}RIP framework should be big enough to accommodate the 1588 message and the biggest RT message and small enough to meet the deadlines. As seen in the Chapter 5, the slot duration depends on many parameters and needed to be experimentally determined. The implementation in this thesis is realized as generic as possible. Still, the slot duration to be used will differ for different applications and hardware.

In real life, the user of D^{3}RIP framework would not want to deal with so many experiments and calculations. To ease the use of this framework, the experiments and calculations made in the Chapter 4 and Chapter 5, could be realized in the implementation of the IL. The user will only give the maximum RT message to be used as an input. Before starting the TDMA structure, IL implementation will conduct the experiments and make the calculations for the *dS lot* and *rem* durations, as done in the Chapter 5.

6.2.3 Token Passing Based Solution

In the current situation, the IL, divides the time into fixed-length time slots, called transmission window. The length of these slots is determined by considering the longest RT message, since in [6], it is assumed that all messages fit into the transmission window, i.e.,

$$m.length < dS\,lot - rem$$

for all messages where, *rem* is the time needed for the protocol related computations of the IL and the CL.

This is the theoretical limit for the transmission window. In practice, there are many delay parameters that should be considered in the calculation of the transmission window, like timer accuracy, synchronization accuracy, propagation delay, etc. These delay parameters should be considered with their maximum value to calculate the time slot size to handle the worst case scenarios. To obtain these maxima values, many experiments should be conducted. Since some of these parameters, like timer accuracy, do not have theoretical maximum values, the experimentally obtained maximums would not be 100%. Therefore some guard values should be added.

As an improvement, instead of calculating the theoretical slot size and dividing the time into

transmission windows via timers, using message reception to start the next transmission is suggested. In other words, instead of implementing a TDMA based medium access, token passing based solution could be used.

This suggestion depends on the broadcast transmission of messages and distributiveness of the D^3RIP framework protocol. In D^3RIP framework, all nodes have the same scheduling of the whole system, i.e., all nodes know which node owns the next slot. Since D^3RIP framework uses a shared-medium broadcast channel, like Ethernet, messages are received by all nodes. So, when the nodes receive the message, the owning node will send the next message after waiting some guard period, and the other nodes will wait it.

The reason of the guard period is that the nodes will not receive the messages at the same time, because of the propagation delay. The amount of this guard period will be found by exchanging messages between each node pairs, and finding the maximum propagation delay in the system.

The transmission windows obtained by this technique will be more deterministic than the one that uses timer, since both guard period (i.e. propagation delay) and the transmission time are constant values that depend on physical properties of the medium and are theoretically calculable. The only periods that will vary are the time needed for the message to be put on the medium and to be received from the medium, which is also a delay parameter in the current solution.

The only point that should be cautiously considered is that if there is no nRT packets for one node, when it is scheduled a nRT slot. In this case, the total system will wait for a nRT packet to arrive. Therefore, the whole system would stop. To solve this problem, the nodes could send dummy messages even if there is no nRT messages when they are assigned an nRT slot.

REFERENCES

[1] A. S. Tanenbaum, *Computernetwerken (Computer Networks)*. No. ISBN 90-430-0698-X, Pearson Prentice Hill, (fourth edition ed.) ed., 2003.

[2] B. H. Jansen, D., "Real-time ethernet the ethercat solution," *Computing & Control Engineering Journal*, vol. 15 , Issue:1, pp. 16 – 21, 2004.

[3] E. Schemm, "Sercos to link with ethernet for its third generation," *Computing & Control Engineering Journal*, vol. 15 , Issue:2, pp. 30 – 33, 2004.

[4] M. Felser, "Real-time ethernet - industry prospective," *Proceedings of the IEEE*, vol. 93 Issue:6, pp. 1118 – 1129, 2005.

[5] C. S. H. Joon Heo, "A security mechanism for automation control in plc-based networks," in *Power Line Communications and Its Applications, 2007. ISPLC '07. IEEE International Symposium on*, 2007.

[6] K. Schmidt and E. G. Schmidt, "Distributed real-time protocols for industrial control systems: Framework and examples," *Transactions on Parallel and Distributed Systems*, 2011.

[7] J.-P. Thomesse, "Fieldbus technology in industrial automation," *Proceedings of the IEEE*, vol. 93 , Issue:6, pp. 1073 – 1101, 2005.

[8] R. P. Maestro, J.A., "Energy efficiency in industrial ethernet: The case of powerlink," *Industrial Electronics, IEEE Transactions on*, vol. 57 , Issue:8, pp. 2896 – 2903, 2010.

[9] J.-d. Decotignie, "The many faces of industrial ethernet [past and present]," *Industrial Electronics Magazine, IEEE*, vol. 3 , Issue:1, pp. 8 – 19, 2009.

[10] M. Schwarz, "Implementation of a ttp/c cluster based on commercial gigabit ethernet components," Master's thesis, Vienna University of Technology, Vienna, Austria, 2011.

[11] S. J. Mullender, *Distributed Systems*. Addison-Wesley, 1993.

[12] "1588tm-2002 standard for a precision clock synchronization protocol for networked measurement and control systems," 2002.

[13] U. Turan, "Implementation and evaluation of a new protocol for industrial communication networks," Master's thesis, Middle East Technical University, 2011.

[14] D. P. Bovet and M. Cesati, *Understanding the Linux Kernel*, ch. CHAPTER 6 - Timing Measurements, pp. p193–216. O'Reilly Media, Inc., 2006.

[15] H. Sivencrona, L. A. Johannson, and V. Claesson, "A novel bit-oriented communication concept for distributed real-time systems,," in *Proc. Third International Conf. Control and Diagnostics in Automotive Applications*, 2001.

[16] “Ieee. carrier sense multiple access with collision detection (csma/cd) access method and physical layer specifications. ansi/ieee std 802.3-2000 edition (iso/iec 8802-3:2000(e)).”

[17] A. R. Jonathan Corbet and G. Kroah-Hartma, *Linux Device Drivers*, ch. CHAPTER 17 - Network Drivers, pp. p497–545. O’Reilly Media, Inc., third edition ed., 2005.

[18] Intel, *Intel® 82574 GbE Controller Family Datasheet*. Intel.

[19] IEEE, “”the ieee’s list of public ethernet type assignments” available: http://standards.ieee.org/develop/regauth/ethertype/eth.txt. last accessed 18th july 2011.,” October 2004.

[20] A. R. Jonathan Corbet and G. Kroah-Hartma, *Linux Device Drivers*, ch. CHAPTER 3 - Char Drivers, pp. p42–72. O’Reilly Media, Inc., third edition ed., 2005.

[21] A. R. Jonathan Corbet and G. Kroah-Hartma, *Linux Device Drivers*, ch. CHAPTER 2 - Building and Running Modules, pp. p15–41. O’Reilly Media, Inc., 2005.

[22] D. P. Bovet and M. Cesati, *Understanding the Linux Kernel*, ch. CHAPTER 17 - System Calls, pp. p303–317. O’Reilly Media, Inc., 2006.

[23] “Free electrons. (2007). uaccess.c source code. available: http://lxr.free-electrons.com/source/arch/um/kernel/skas/uaccess.c. last accessed 30th jan 2011..”

[24] J. S. Gray, *Interprocess Communications in Linux*. Prentice Hall Professional, 2003.

[25] W. Mauere, *Professional Linux Kernel Architecture*, ch. Chapter 6: Device Driver, pp. pg391–473. Wiley Publishing, Inc, 2008.

[26] D. P. Bovet and M. Cesati, *Understanding the Linux Kernel*, ch. Chapter 11: Signals, pp. pg420–456. O’Reilly Media, Inc., 2006.

[27] H. . S. Y. . Kamimura, K. ; Hoshino, “Constant delay queuing for jitter-sensitive iptv distribution on home network,” in *Global Telecommunications Conference, 2008. IEEE GLOBECOM 2008. IEEE*, 2008.

[28] “Ieee 1588 standard for a precision clock synchronization protocol for networked measurement and control systems.[online]. available:http://ieee1588.nist.gov, 2002..”

[29] “Wind river. (2011). the world’s no:1 rtos for embedded real time systems. available: http://www.windriver.com/products/vxworks/. last accessed 19th aug 2011..”

[30] D. Abbott, *Linux for Embedded and Real-Time Applications*, ch. Chapter 9: Linux and Real-time, pp. p147–154. Newnes, 2003.

[31] “Wind river. (2007). rtlinuxfree. available: http://www.rtlinuxfree.com/. last accessed 16th jan 2011..”

[32] “Politecnico di milano - dipartimento di ingegneria aerospaziale. (2006). rtai - real time application interface official website. available: https://www.rtai.org/. last accessed 20th jan 2011..”

[33] “Osadl - open source automation development lab. (2010). osadl project: ”latest stable” rt-preempt realtime linux kernel. available: https://www.osadl.org/latest-stable-realtime.latest-stable-realtime-linux.0.html. last accessed 21 dec 2010..”

[34] “Free electrons. (2004). real-time in embedded linux systems.available: http://free-electrons.com/doc/embedded_linux_realtime.pdf. last accessed 20th dec 2010..”

Appendix A

TIOA DESCRIPTION FOR D^{3}RIP

A.1 TIOA Description For SM

TIOA $SM(dNumber, M)$ where $dNumber \in \mathbb{N}$

Variables X	Actions A
$\texttt{mess}^{\text{d}} \in \mathcal{M}$ (empty)	**input** IL2SM$(m)_i$, $m \in M$,
$\texttt{coll}^{\text{d}} \in \mathbb{B}$ (false)	$1 \leq i \leq dNumber$
$\texttt{next}^{\text{d}} \in \mathbb{R}$ (0)	**output** SM2IL(m), $m \in M$
$\texttt{now}^{\text{a}} \in \mathbb{R}$ (0)	
Tansitions D	
input IL2SM$(m)_i$	**output** SM2IL(m)
effect:	precondition:
if mess is empty	$\texttt{now}^{\text{a}} = \texttt{next}^{\text{d}}$
$\texttt{mess}^{\text{d}} = m$	effect:
$\texttt{next}^{\text{d}} = \texttt{now}^{\text{a}} + m$.length	$m = \texttt{mess}^{\text{d}}$
else	set $\texttt{mess}^{\text{d}}$ empty
$\texttt{coll}^{\text{d}} =$ true	$\texttt{next}^{\text{d}} = 0$
$\texttt{next}^{\text{d}} = 0$	
set $\texttt{mess}^{\text{d}}$ empty	
Tajectories T	
stop when	**evolve**
$\texttt{now}^{\text{a}} = \texttt{next}^{\text{d}} \wedge \texttt{mess}^{\text{d}}$ not empty	$d(\texttt{now}^{\text{a}}) = 1$

A.2 TIOA Description For IL

TIOA $IL_i(dS\,lot, rem, cmp, M, Q, A_{IL})$

Variables X

$\texttt{now}_i^{\text{a}} \in \mathbb{R}$ (0)

$\texttt{next}_i^{\text{d}} \in \mathbb{R}$ ($dS\,lot$)

$\texttt{TxRT}_i^{\text{d}} \in M$ (empty)

$\texttt{TxnRT}_i^{\text{d}} \in Q$ (empty)

$\texttt{RxRT}_i^{\text{d}} \in M$ (empty)

$\texttt{RxnRT}_i^{\text{d}} \in Q$ (empty)

$\texttt{RTIL}_i^{\text{d}} \in \mathbb{B}$ (false)

$\texttt{myIL}_i^{\text{d}} \in \mathbb{B}$ (false)

$\texttt{vIL}_i^{\text{d}} \in A_{\text{IL}}$ (InitV)

$\texttt{reqIL}_i^{\text{d}} \in \mathbb{B}$ (false)

Transitions D

internal UPDATE$_i$

precondition:

$\texttt{now}_i^{\text{a}}$=$\texttt{next}_i^{\text{d}} - rem$

$\texttt{RxRT}_i^{\text{d}}$ empty

effect:

$\texttt{vIL}_i^{\text{d}} =$

$f_{\text{upd}}(\texttt{vIL}_i^{\text{d}}, \texttt{RTIL}_i^{\text{d}})$

$\texttt{reqIL}_i^{\text{d}} = f_{\text{req}}(\texttt{vIL}_i^{\text{d}})$

if $\neg\texttt{reqIL}_i^{\text{d}}$

$\texttt{RTIL}_i^{\text{d}} =$ false

$\texttt{myIL}_i^{\text{d}} =$

$f_{\text{my}}(\texttt{vIL}_i^{\text{d}}, \texttt{RTIL}_i^{\text{d}}, b_2, i)$

$\texttt{next}_i^{\text{d}} = \texttt{next}_i^{\text{d}} + dS\,lot$

output IL2CLRT$(m, \texttt{now}_i^{\text{a}})_i$

precondition:

$\texttt{now}_i^{\text{a}}$=$\texttt{next}_i^{\text{d}} - rem$

$\neg(\texttt{RxRT}_i^{\text{d}}$ empty)

Actions A

input SM2IL$(m), m \in M$

input AP2ILNRT$(m)_i, m \in M$

input CL2ILRT$(b_1, b_2, m)_i$,

$m \in M$, $b_1, b_2 \in \mathbb{B}$

input IL2APNRT$(q)_i, q \in Q$

output IL2CLRT$(m, t)_i$,

$m \in M$, $t \in \mathbb{R}$

output IL2SM$(m)_i$, $m \in M$

internal UPDATE$_i$

output REQRT$(t)_i$, $t \in \mathbb{R}$

input SM2IL(m)

effect:

if $\texttt{RTIL}_i^{\text{d}}$

$\texttt{RxRT}_i^{\text{d}} = m$

else

$\texttt{RxnRT}_i^{\text{d}}$.Push$(m)$

output IL2SM$(m)_i$

precondition:

$(\texttt{now}_i^{\text{a}} = \texttt{next}_i^{\text{d}} - dS\,lot) \wedge$

$\texttt{myIL}_i^{\text{d}}$

$(\neg(\texttt{TxRT}_i^{\text{d}}$ empty$) \wedge$

$\texttt{RTIL}_i^{\text{d}}) \vee (\neg\texttt{RTIL}_i^{\text{d}} \wedge$

$\neg(\texttt{TxnRT}_i^{\text{d}}$.Top empty$))$

effect:

if $\texttt{RTIL}_i^{\text{d}}$

set $m = \texttt{TxRT}_i^{\text{d}}$

set $\texttt{TxRT}_i^{\text{d}}$ empty

effect:

set $m = \texttt{RxRT}_i^{\text{d}}$

set $\texttt{RxRT}_i^{\text{d}}$ empty

output REQRT($\texttt{now}_i^{\text{a}}$)$_i$

precondition:

$\texttt{reqIL}_i^{\text{d}}$ = true

$\texttt{now}_i^{\text{a}} = \texttt{next}_i^{\text{d}} - dS\,lot -$

$rem + cmp$

effect:

$\texttt{reqIL}_i^{\text{d}}$ = false

input IL2APNRT($\texttt{RxnRT}_i^{\text{d}}$)$_i$

effect:

set $\texttt{RxnRT}_i$ empty

if $\neg\texttt{RTIL}_i^{\text{d}}$

set $m = \texttt{TxnRT}_i^{\text{d}}$.Top

$\texttt{TxnRT}_i^{\text{d}}$.Pop

$\texttt{myIL}_i^{\text{d}}$ = **false**

input CL2ILRT(b_1, b_2, m)$_i$

effect:

$\texttt{RTIL}_i^{\text{d}} = f_{\text{RT}}(\texttt{vIL}_i^{\text{d}}, b_1)$

$\texttt{myIL}_i^{\text{d}} =$

$f_{\text{my}}(\texttt{vIL}_i^{\text{d}}, \texttt{RTIL}_i^{\text{d}}, b_2, i)$

$\texttt{TxRT}_i^{\text{d}} = m$

input AP2ILNRT(m)$_i$

effect:

$\texttt{TxnRT}_i^{\text{d}}$.Push($m$)

Trajectories T

stop when

$\texttt{now}_i^{\text{a}} = \texttt{next}_i^{\text{d}} - dS\,lot \wedge \texttt{myIL}_i^{\text{d}}$

$\texttt{now}_i^{\text{a}} = \texttt{next}_i^{\text{d}} - rem$

$(\texttt{now}_i^{\text{a}} = \texttt{next}_i^{\text{d}} - dS\,lot - rem + cmp) \wedge \texttt{reqIL}_i^{\text{d}}$

evolve

$d(\texttt{now}_i^{\text{a}}) = 1$

Table A.1: Protocol Specific Functions for IL

	RAIL	**TSIL**
$f_{\text{upd}}(\texttt{vIL}_i^{\text{d}}, \texttt{RTIL}_i^{\text{d}})$	($\texttt{vIL}_i^{\text{d}}$.cnt + 1) mod $\texttt{vIL}_i^{\text{d}}$.cyc	$\texttt{vIL}_i^{\text{d}}$.cnt if $\texttt{RTIL}_i^{\text{d}}$ ($\texttt{vIL}_i^{\text{d}}$.cnt + 1) mod $\texttt{vIL}_i^{\text{d}}$.cyc otherwise
$f_{\text{req}}(\texttt{vIL}_i^{\text{d}})$	true if $\texttt{vIL}_i^{\text{d}}$.cnt $\in$ $\texttt{vIL}_i^{\text{d}}$.RTset false otherwise	true
$f_{\text{RT}}(\texttt{vIL}_i^{\text{d}}, b_1)$	true if $\texttt{vIL}_i^{\text{d}}$.cnt $\in$ $\texttt{vIL}_i^{\text{d}}$.RTset $\wedge b_1$ = true false otherwise	true if b_1 = true false otherwise
$f_{\text{my}}(\texttt{vIL}_i^{\text{d}}, \texttt{RTIL}_i^{\text{d}}, b_2, i)$	b_2 if $\texttt{RTIL}_i^{\text{d}}$ true if $\neg\texttt{RTIL}_i^{\text{d}} \wedge$ $\texttt{vIL}_i^{\text{d}}$.cnt $\in$ $\texttt{vIL}_i^{\text{d}}$.nRTSet false otherwise	b_2 if $\texttt{RTIL}_i^{\text{d}}$ true if $\neg\texttt{RTIL}_i^{\text{d}} \wedge$ $\texttt{vIL}_i^{\text{d}}$.cnt $\in$ $\texttt{vIL}_i^{\text{d}}$.nRTSet false otherwise

A.3 TIOA Description For CL

TIOA $CL_i(del_i, M, Q, V, A_{CL}), del_i \in \mathbb{R}$

Variables X

$\mathtt{send}_i^{\mathrm{a}} \in \mathbb{R}$ (del_i)

$\mathtt{Tx}_i^{\mathrm{d}} \in V$ (empty)

$\mathtt{Rx}_i^{\mathrm{d}} \in Q$ (empty)

$\mathtt{RTCL}_i^{\mathrm{d}} \in \mathbb{B}$ (false)

$\mathtt{myCL}_i^{\mathrm{d}} \in \mathbb{B}$ (false)

$\mathtt{ch}_i^{\mathrm{d}} \in \mathbb{N}$ (0)

$\mathtt{reqCL}_i^{\mathrm{d}} \in \mathbb{B}$ (false)

$\mathtt{vCL}_i^{\mathrm{d}} \in A_{\mathrm{CL}}$ $(InitCL)$

Transitions D

input AP2CL$(dat, p, ch)_i$

effect:

$\mathtt{Tx}_i^{\mathrm{d}}[ch].\mathtt{data} = dat$

$\mathtt{Tx}_i^{\mathrm{d}}[ch].\mathtt{par} = p$

input REQRT$(t)_i$

effect:

$\mathtt{RTCL}_i^{\mathrm{d}} =$

$g_{RT}(\mathtt{vCL}_i^{\mathrm{d}}, \mathtt{RTCL}_i^{\mathrm{d}}, t)$

$(\mathtt{myCL}_i^{\mathrm{d}}, \mathtt{ch}_i^{\mathrm{d}}) =$

$g_{my}(\mathtt{vCL}_i^{\mathrm{d}}, \mathtt{RTCL}_i^{\mathrm{d}}, t, i)$

$\mathtt{send}_i^{\mathrm{a}} = 0$

$\mathtt{reqCL}_i^{\mathrm{d}} =$ true

input CL2AP$(\mathtt{Rx}_i^{\mathrm{d}})_i$

effect:

set $\mathtt{Rx}_i^{\mathrm{d}}$ empty

Trajectories T

stop when

$(\mathtt{send}_i^{\mathrm{a}} = del_i) \wedge \mathtt{reqCL}_i$

Actions A

input AP2CL$(dat, p, ch)_i$, $dat \in M$.data, $p \in M$.par, $ch \in \mathbb{N}$

input CL2AP$(q)_i$, $q \in Q$

input IL2CLRT$(m, t)_i$, $m \in M$, $t \in \mathbb{R}$

input REQRT$(t)_i$, $t \in \mathbb{R}$

output

CL2ILRT$(\mathtt{RTCL}_i^{\mathrm{d}}, \mathtt{myCL}_i^{\mathrm{d}}, m)_i$

input IL2CLRT$(m, t)_i$

effect:

$\mathtt{Rx}_i^{\mathrm{d}}$.Push$(m)$

$\mathtt{vCL}_i^{\mathrm{d}} = g_{\mathrm{upd}}(\mathtt{vCL}_i^{\mathrm{d}},$

$m.\mathtt{par}, t)$

output

CL2ILRT$(\mathtt{RTCL}_i^{\mathrm{d}}, \mathtt{myCL}_i^{\mathrm{d}}, m)_i$

precondition:

$\mathtt{reqCL}_i^{\mathrm{d}} \wedge (\mathtt{send}_i^{\mathrm{a}} \leq del_i)$

effect:

if $\mathtt{myCL}_i^{\mathrm{d}}$

set $m = \mathtt{Tx}_i^{\mathrm{d}}[\mathtt{ch}_i^{\mathrm{d}}]$

set $\mathtt{Tx}_i^{\mathrm{d}}[\mathtt{ch}_i^{\mathrm{d}}]$ empty

else

set m empty

$\mathtt{reqCL}_i^{\mathrm{d}} =$ false

evolve

$d(\mathtt{send}_i^{\mathrm{a}}) = 1$

Table A.2: Protocol Specific Functions for CL

	DART	**URT**
$g_{\text{upd}}(\text{vCL}_i^{\text{d}}, \text{m.par}, t))$	$(\text{vCL}_i^{\text{d}}.\text{cnt} + 1) \mod \text{vCL}_i^{\text{d}}.\text{cyc}$	$(\text{vCL}_i^{\text{d}}.\text{cnt} + 1) \mod \text{vCL}_i^{\text{d}}.\text{cyc}$
$g_{\text{RT}}(\text{vCL}_i^{\text{d}}, \text{RTCL}_i^{\text{d}}, t))$	true if $\text{vCL}_i^{\text{d}}.\text{cnt} \in \text{vCL}_i^{\text{d}}.\text{alloc}[k].\text{slots}$ for some k false otherwise	true if $\text{vCL}_i.\text{PQ}.\text{Top}.eT \leq t$ false otherwise
$g_{\text{my}}(\text{vCL}_i^{\text{d}}, \text{RTCL}_i, t, i)$	(true,c) if $\text{vCL}_i^{\text{d}}.\text{cnt} \in \text{vCL}_i^{\text{d}}.\text{alloc}[k].\text{slots}$ $\wedge \text{vCL}_i^{\text{d}}.\text{alloc}[k].\text{used} = (i, c)$ (false,0) otherwise	(true,a) if $\text{PQ}_i.\text{Top}.b = i \wedge \text{RT}_i$ $= \text{true} \wedge \text{PQ}_i.\text{Top}.c = a$ (false,0) otherwise

Appendix B

OPERATING SYSTEM CHOICE

B.1 Introduction

Since D^3RIP framework is a real-time communication protocol, the choice of the operating system, which this protocol will run on, is crucial. The primary characteristic of real-time communications is that the messages should be delivered within a predetermined period, i.e., they should meet their deadlines. So, the operating system that provides services to the protocol also should have RT characteristics.

In academic world, Linux become a widely used operating system, as its advantages, like being open source, having community support, and being free; cannot be obtained from any other operating system.

Apart from these advantages, Linux has a crucial deficiency: It is not real-time. Linux, is developed as a general-purpose operating system, and just like any other general-purpose operating system, it is tuned to maximize average throughput even at the expense of latency. Real-time operating systems, on the other hand, attempt to minimize, and place an upper bound on, latency, sometimes at the expense of average throughput.

According to Doug Abbott [30], standard Linux is not suitable for real-time use for five reasons:

- Coarse-grained Synchronization → Kernel system calls are not preemptive
- Paging → Swapping pages in and out of virtual memory is unbounded
- Fairness in Scheduling → Low priority process may run even though a higher-priority process is ready.

- Request Reordering

- Batching

For resolving these deficiencies, some modifications are imported into new kernel versions.

For example, to improve latencies, new scheduling policies; SCHED_FIFO and SCHED_RR, are proposed instead of the default Linux time-sharing scheduling policy, SCHED_OTHER, which uses a fairness algorithm and gives all processes using this policy the lowest priority.

Another improvement in the Linux kernel is the preemption improvement. With version 2.5.4-pre6, the preemption improvements, using spinlocks to support symmetric multi-processing (SMP), are merged into the main kernel tree. With this improvement, maximum process latency, which is in the order of tens of milliseconds for a standard kernel, reduced to one or two milliseconds.

Still, these improvements do not provide a fully preemptive kernel, so they only help us to achieve soft real-time behavior. To improve Linux's performance to near hard real-time performance, more drastical changes should be applied to the kernel.

There are three famous ways to achieve real-time performances from Linux:

1. RTLinux [31]

2. RTAI [32]

3. RT-preempt patch [33]

B.2 RTLinux

RTLinux, become a commercial product and only old versions can be downloaded and used freely. The last free version of RTLinux is 3.1 and the latest Linux kernel that it supports is 2.6.9. Therefore, RTLinux is no longer a preferred real-time Linux support in academic area.

B.3 RTAI

Being a community product and free makes RTAI a better option than RTLinux. Nevertheless, of the late years, new versions are not being released frequently (RTAI 3.7.1 released in 06/15/2009 - RTAI 3.8 released in 02/16/2010) [32], and they only support i386 seriously. Also, RTAI's lack of stability and industrial maturity (since the development is driven by the immediate needs of its maintainer) and using different Application Programming Interface (API) from the traditional Linux API (RTAI API) discourages the Linux users, who want to obtain RT support from Linux.

B.4 RT-preempt patch

As desire for getting real-time performance from Linux arise, many real-time patches have been released. These patches got into the basic Linux kernel tree with time. Some of these improvements and the Linux kernel versions that include these modifications are given below:

- High-resolution timer → 2.6.24
- Preemptive Read-Copy Update → 2.6.25
- IRQ Threads → 2.6.30
- Raw Spinlock Annotation → 2.6.33

Open Source Automation Development Lab's (OSADL)[33] Realtime Linux project contribute to these improvements by supporting the evolution of the realtime preempt patch maintained by Ingo Molnar, and Thomas Gleixner. The advantages of these patches is that they are still free and following current progresses in the Linux kernel. With the latest real-time patch [33], complete (real-time) preemption had achieved, which overcome a critical problem. Still there are some shortages to obtain hard real-time performance, such as [34]:

- Though being preemptible, kernel services, like memory allocation, do not have a guaranteed latency yet.
- Kernel drivers are not developed for real-time constraints.

- There are binary-only drivers which somehow should be recompiled for RT preempt.

B.5 Conclusion

Being a widely used operating system, Linux draw attention of the areas that use embedded technology. The biggest disadvantage of Linux is that it is developed as a general-purpose operating system and modifications should be made in order to use in real-time embedded areas. Latest real-time patches, greatly improved the performance of Linux. Also following latest advancements in Linux kernel closely makes these patches a good candidate for obtaining a Linux with nearly hard real-time performance.