IMPLEMENTATION AND EVALUATION OF THE DYNAMIC DISTRIBUTED REAL TIME INDUSTRIAL PROTOCOL (D^2RIP)

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

BY

ADEM KAYA

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

SEPTEMBER 2013

Approval of the thesis:

IMPLEMENTATION AND EVALUATION OF THE DYNAMIC DISTRIBUTED REAL TIME INDUSTRIAL PROTOCOL (D^2RIP)

submitted by **ADEM KAYA** 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. Gönül Turhan Sayan
Head of Department, **Electrical and Electronics Engineering** ___________

Assoc. Prof. Dr. Şenan Ece Schmidt
Supervisor, **Electrical and Electronics Engineering Dept., METU** ___________

Assoc. Prof. Dr. Klaus Werner Schmidt
Co-Supervisor, **Mechatronics Engineering Dept., Çankaya University** ___________

Examining Committee Members:

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

Assoc. Prof. Dr. Şenan Ece Schmidt
Electrical and Electronics Engineering Dept., METU ____________________

Assoc. Prof. Dr. Cüneyt Bazlamaççı
Electrical and Electronics Engineering Dept., METU ____________________

Assoc. Prof. Dr. Halit Oğuztüzün
Computer Engineering Dept., METU ____________________

Yusuf Bora Kartal (M.Sc.)
Engineer, ASELSAN ____________________

Date: **02.09.2013**

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 : Adem KAYA

Signature :

ABSTRACT

IMPLEMENTATION AND EVALUATION OF THE
DYNAMIC DISTRIBUTED REAL TIME INDUSTRIAL PROTOCOL
(D²RIP)

Kaya, Adem
M. S., Department of Electrical and Electronics Engineering
Supervisor : Assoc. Prof. Dr. Şenan Ece Schmidt
Co-Supervisor : Assoc. Prof. Dr. Klaus Werner Schmidt

September 2013, 81 pages

The contemporary large-scale and complex industrial control systems such as manufacturing systems, power plants or chemical processes are realized as distributed systems. Since different controller nodes are usually physically distributed, their coordination and information exchange is commonly realized via industrial communication networks (ICNs). In the last decade, there is an ongoing research effort in both academic and industrial fields to employ Ethernet for industrial communications due to its wide acceptance and use in home and office networks. Although the conventional Ethernet technology is low-cost and very high-speed its nondeterministic behavior does not support real-time traffic.

In this thesis we present the design, implementation and evaluation of the novel ICN protocol D²RIP (Dynamic Distributed Real-time Industrial Communication Protocol) that was proposed in previous work. D²RIP is a fully distributed protocol over shared-medium Ethernet with COTS (Commercial Off-The-Shelf) hardware and provides real-time message delivery guarantees, supports non-real-time traffic. As a distinctive feature in comparison to other ICNs over Ethernet that only support static allocation of real-time and non-real-time bandwidth, D²RIP allows for dynamic allocation of the network capacity among the participating nodes by exploiting knowledge about the deterministic system behavior of industrial systems.

Keywords: Real-time Ethernet, Industrial Communication Protocols, Shared Medium, Dynamic Bandwidth Allocation

ÖZ

DİNAMİK DAĞITILMIŞ GERÇEK-ZAMANLI ENDÜSTRİYEL PROTOKOLÜN (D^2GEP)
GERÇEKLEŞTİRİLMESİ VE DEĞERLENDİRİLMESİ

Kaya, Adem
Yüksek Lisans, Elektrik Elektronik Mühendisliği Bölümü
Tez Yöneticisi : Doç. Dr. Şenan Ece Schmidt
Ortak Tez Yöneticisi : Doç. Dr. Klaus Werner Schmidt

Eylül 2013, 81 sayfa

Günümüz otomasyon sistemlerinde kontrol uygulamaları pek çok sayısal cihaz üzerinde dağıtılmış gömülü sistemler olarak gerçekleştirilmektedir. Bu cihazların koordinasyonu ve haberleşmesi endüstriyel haberleşme ağları üzerinden yapılmaktadır. Günümüzde ev ve ofis ağlarında çok yaygın olarak kullanılan fakat gerçek zamanlı haberleşmeyi desteklemeyen standart Ethernet teknolojisinin, endüstriyel haberleşme ağlarında kullanılması önemli bir araştırma konusudur.

Bu tezde, iki katmanlı gerçek zamanlı endüstriyel haberleşme mimarisinin, D^2RIP (Dynamic Distributed Real-time Industrial Communication Protocol) (Dinamik Dağıtık Gerçek Zamanlı Endüstriyel İletişim Protokolü (D^2GEP)) tasarımı, uygulaması ve geliştirilmesi anlatılacaktır. D^2GEP mimarisi tamimiyle paylaşımlı ortam Ethernet haberleşmesi üzerine kurulmuş olup, gerçek zamanlı mesaj iletimini sağladığı gibi gerçek olmayan mesaj trafiğini de desteklemektedir. Bu mimaride ağ kaynakları anlık gerçek zamanlı haberleşme ihtiyaçlarına göre dinamik bir biçimde düğümlere dağıtılmakta ve artan ağ kapasitesi gerçek zamanlı olmayan trafik için kullanılmaktadır.

Anahtar Kelimeler: Gerçek Zamanlı Ethernet, Endüstriyel Haberleşme Protokolü, Paylaşımlı Ortam, Dinamik Bant Genişliği Ayrımı

To My Parents

ACKNOWLEDGMENTS

I would like to express my sincere gratitude to my supervisor Assoc. Prof. Dr. Şenan Ece Schmidt and my co-supervisor Assoc. Prof. Dr. Klaus Werner Schmidt for their guidance, advice, criticism, encouragement and insight throughout the completion of the thesis. Besides, I like to thank them providing me the opportunity to work on such an important issue.

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

I would like to special reference for my employer, ASELSAN for encouraging me to complete my studies in the university.

Finally, I am grateful to my parents for their continuous support and encouragements.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS AND ACRONYMS

AL	Application Layer
API	Application Programming Interface
BEB	Binary Exponential Back Off
C	Conveyor
CIM	Computer Integrated Manufacturing
CIP	Common Industrial Protocol
CL	Coordination Layer
COTS	Commercial Off-The-Shelf
CRC	Cyclic Redundancy Check
CSMA/CD	Carrier Sense Multiple Access with Collision Detection
CSME	Communication Schedule Management Entity
D^{2}RIP	Dynamic Distributed Real-time Industrial Communication Protocol
DT	Deadline Time
DP	Dependability Plane
EPA	Ethernet for Plant Automation
EPL	Ethernet Powerlink
ET	Eligibility Time
FTP	File Transfer Protocol
HPET	High Precision Event Timer
HTTP	Hyper-Text Transfer Protocol
ICN	Industrial Communication Networks
IL	Interface Layer
IOCTL	Input/Output Control
IRQ	Interrupt Request
MAC	Medium Access Control
MII	Medium Independent Interface
NIC	Network Interface Card
NTP	Network Time Protocol
nRT	Non-Real-Time
OS	Operating System

PC	Personal Computer
PD	Painting Device
PI	Proportional-Integral
PHC	PTP Hardware Clock
PLC	Programmable Logic Controller
PPS	Pulse Per Second
PTP	Precision-Time Protocol
R	Robot
RT	Real-Time
RTC	Real Time Clock
RTOS	Real-Time Operating System
SM	Shared Medium
TCNet	Time Critical Control Network
TSC	Time Stamp Counter
TDMA	Time Division Multiple Access
VTPE	The Virtual Token Passing

CHAPTER 1

INTRODUCTION

The devices and components of contemporary automation and manufacturing systems are computer controlled to realize complicated tasks. These tasks are implemented as distributed applications where the participating devices are required to coordinate their operation. To this end, Industrial Communication Networks (ICNs) are employed to transport the messages that are generated by industrial applications. Due to the specific features of the industrial applications, ICNs have to fulfill various requirements [1] [2].

First and foremost, the *timely delivery* of messages has to be guaranteed. In particular, real-time (RT) messages that are either *periodic* (e.g., from position control) or *sporadic* (e.g., from limit switches) have to be delivered with small delays. In addition to RT messages, non-real-time (nRT) messages related to time-uncritical decisions of the controllers [3] and data communications for diagnosis or maintenance [4] [5] should also be delivered. Second, it has to be respected that the equipment and their arrangements such as cabling in the industrial environment are often designed to be used over a number of years. Hence, *efficient use of resources* is important for the ICN to enable the adoption of possible new applications and the extension of the system without replacing the network infrastructure. Third, the support of RT communication as well as the time stamping of diagnostic messages requires accurate synchronization among distributed system components [1]. Finally, the compatibility with COTS Ethernet hardware is highly desirable.

A number of different ICN standards are developed over the years. Initially these standards were proprietary and specific to the application which lacked compatibility among each other. This both hindered integrating different standards to construct large-scale heterogeneous systems and the continuous development of each proprietary standard [6]. The widely accepted solution to these problems is employing *Ethernet* which is the de-facto standard for home and office networks [7]. Ethernet has low-cost hardware and provides a raw bandwidth of 10/100/1000Mbps. However, the medium access of conventional shared medium Ethernet (IEEE 802.3) is nondeterministic which does not comply with the operation requirements of industrial applications. Hence, it is an important research problem to employ Ethernet as an ICN.

The main motivation for Ethernet-based ICN is its wide adoption and low cost hardware. Hence, altering the Ethernet operation and the hardware to achieve the desired determinism is not a viable solution [1]. Another approach is *traffic shaping* with the aim of improving the average message delivery times. However, most control applications require hard worst-case real-time guarantees instead of probabilistic bounds. The conventional Ethernet is a shared medium protocol with a probabilistic collision resolution mechanism. *Switched Ethernet* eliminates the collisions by point to point communication [8]. However, in addition to the requirement of re-cabling and installing switches, the messages that are transmitted at the same time are queued in buffers and certain buffer management capabilities are required to achieve real-time delivery [1]. Few protocols support on-demand resource allocation to some degree by token rotation or master slave arrangements [9].

In this thesis, we review the design, implementation and evaluation of the novel Ethernet-based ICN protocol D^2RIP – Dynamic Distributed Real-time Industrial Communication Protocol. D^2RIP is developed to be fully distributed with dynamic bandwidth allocation and operates on conventional shared-medium Ethernet with COTS (Commercial Off-The-Shelf) Ethernet hardware. Both RT traffic and nRT traffic are supported. Regarding the protocol design, D^2RIP is a two-layer protocol stack on top of shared-medium Ethernet. Its lower *Interface Layer* (IL) implements a time-slotted medium access based on accurate time synchronization. In this work, the IEEE 1588 precision clock synchronization protocol for networked measurement and control systems [10] is used for this purpose. In order to avoid collisions, it is further required to allocate each time slot to a unique node. This task is achieved by our upper *Coordination Layer* (CL). The CL of D^2RIP dynamically determines the node that is allowed to transmit in each current slot by a distributed computation on all nodes. Hereby, information from the control application is used in order to dynamically adjust the supplied bandwidth depending on the respective communication requirements.

Although the formulation of D^2RIP in the framework of timed input/output automata [11] in D^2RIP's earlier work [12] ensures collision free transmission on shared-medium Ethernet and the requirements of ICN as discussed above, it remains to evaluate the performance of D^2RIP in a practical application. In this respect, the contributions of this thesis can be listed as follows:

- Determining appropriate data structures for the implementation of D^2RIP according to the general formal framework in [12]. Here we would like to note that D^2RIP is one realization and it is possible to develop possible other protocol stacks according to the framework.

- Implementing the two layer D^2RIP stack on COTS Ethernet hardware and an open source real-time operating system. We provide the implementation details including the time-slotted operation, giving selective medium access to RT and nRT traffic. Our description includes a method for fragmentation and reassembly of long nRT packets.

- We present solutions for the challenges we encountered in achieving precise real-time response on a platform that is not custom designed. Our D^2RIP implementation is entirely portable and works on PCs as well as embedded industrial controller devices such as programmable logic controllers (PLCs).

- A comprehensive experimental evaluation of D²RIP. The experiments are run on an entirely realistic hardware setup with four controller nodes and a simulated manufacturing cell. Our findings show that D²RIP supports an effective RT bandwidth of 16.1 Mbit/s which is similar to comparable protocols. However, D²RIP uses this effective bandwidth more efficiently due to the dynamic bandwidth allocation. In addition, we obtain the average of clock synchronization accuracies in the order of 1.5 µs and max support delivery times below 5 ms. Regarding nRT traffic, we observe that short packets are not efficiently supported by D²RIP due to the time-slotted medium access.

Earlier implementations of D²RIP were implemented and presented in [13] [14] [15]. This thesis benefits from the experiences gained during these studies and re-implements D²RIP.

The remainder of this thesis is organized as follows. We present an overview of Ethernet based ICN protocols together with the requirement definitions and related previous work in Chapter 2. We then introduce the D²RIP stack with its data structure that is derived from the framework by [12] in Chapter 3. Our representation includes quantitative metrics for the protocol evaluation and a practical control application example in order to demonstrate the performance of D²RIP according to the defined metrics. Chapter 4 gives essential implementation details of our D²RIP implementation and shows the differences between current D²RIP's work with the previous implementation. A comprehensive experimental evaluation of D²RIP in a real hardware set-up with 4 controller nodes and a plant simulator is performed in Chapter 5. Finally our conclusions and directions for future work are discussed in Chapter 6.

CHAPTER 2

ETHERNET-BASED INDUSTRIAL COMMUNICATION: REQUIREMENTS AND RELATED WORK

Computer Integrated Manufacturing (CIM) systems implement control systems as a distributed computer application running on a number of intelligent nodes (sensors, actuators, programmable logic controllers – PLCs, industrial PCs, etc.) which are equipped with microprocessors and exchange information over an *industrial communication network*.

Different types of communications such as *device level* communications, *system level* communications or *diagnostic/higher level* communications take place in CIM systems [3]. Device level communication involves sensors and actuators that are attached at the hardware level of a CIM. Sensors periodically generate sampled data and usually require data transmission before a certain deadline. Similarly, device level controllers send computed control commands to the actuators while fulfilling real-time constraints. In addition, system level controllers exchange data among each other to coordinate device level components. This communication is also real-time and often event-triggered. Furthermore, diagnostic data and relevant system management information is carried over the industrial communication network mostly as non-real-time traffic. The amount of communicated data can be large and interface to other networks such as other plants or even the Internet for remote applications can be required at this level.

The most commonly used technology to achieve an overall industrial communication architecture across all system levels is Ethernet which constitutes the dominating network technology in home and office environments because of its high-speed, simplicity and low cost. The first Ethernet standard was issued as IEEE 802.3 in 1985 [6] to run on shared medium where simultaneous transmissions result in collisions and loss of data. The network nodes encapsulate their data payloads in Ethernet frames and listen to the channel before starting the transmission to decrease the number of collisions (Carrier Sense Multiple Access with Collision Detection (CSMA/CD)). However, collisions are still possible because of the propagation delay on the channel between the start of a frame and the time it is sensed by other nodes. In such cases, the lost frames are retransmitted after a time interval of random length. Although this operation is very suitable for home and office applications, where stringent timing constraints are not essential, it does not allow for the deterministic timing required in industrial applications [6]. Hence, the literature provides a significant number of studies that are concerned with equipping standard Ethernet with real-time features so as to enable its deployment for industrial communications [16] [1] [7].

In this chapter, we give a brief overview of the efforts that are most related to the industrial Ethernet protocol presented in this thesis. To this end, Section 2.1 points out basic requirements for industrial communication and Section 2.2 describes several realizations of industrial Ethernet protocols. A discussion of the existing literature in relation to our proposed protocol is given in Section 2.3.

2.1 Requirements

The most important requirements and performance indicators for real-time industrial Ethernet comprise the *real-time (RT) delivery of data*, *support of precise time synchronization*, *support of non-real-time (nRT) traffic*, *compatibility to standard hardware* and *configurability and extendibility* [1].

2.1.1 Real-time Delivery of Data

The *delivery time* of a message is defined *as the difference between the time the message is generated at the application layer of the sender node and the time it is received at the application layer of the receiver node*. The timing requirements of the device level and supervisory control level communications as described above are application dependent. Applications that involve human operators such as process automation and building control requires delivery times in the order of 100 ms. Process control applications such as tooling machine control systems with PLCs require delivery times below 10 ms. Motion control applications such as synchronizing several axes over a network require a delivery time less than 1 ms.

2.1.2 Synchronization Support

The RT response from an industrial network requires a synchronization mechanism to establish a common notion of time among the distributed nodes of the network. *Time synchronization accuracy is the maximum deviation between any two node clocks and non-time-based synchronization accuracy is the maximum jitter of the cyclic behavior of any two nodes* [1]. The time synchronization accuracy affects the guard times of the sent messages and hence the delivery time. Supporting a large number of nodes or demanding applications such as motion control requires sub-microsecond accuracy. While some industrial network implementations rely on their own synchronization mechanisms, others adopt standardized protocols. We provide an overview of the synchronization in RT industrial communication networks in Section 2.2.3.

2.1.3 Non-real-time Traffic Support

The remote management of control processes or factories over web is a desired feature for industrial communication. To this end, it should be possible to carry the high-level management and diagnostics traffic as standard HTTP web traffic or FTP file transfer using TCP/UDP/IP protocols. It is important to note that such nRT traffic has to be transmitted over the industrial communication network without any performance degradation of the RT traffic.

2.1.4 Compatibility

One of the most attractive features of Ethernet is its low cost hardware and software interface. Hence, it is important to maintain compatibility to enable real-time Ethernet implementations with COTS (Commercial Off-The-Shelf) components. Furthermore, compliance with the existing TCP/IP stack is required for enabling the use of popular protocols such as Hyper-Text Transfer Protocol (HTTP) or File Transfer Protocol (FTP). Such protocols are employed in remote management over Internet and for using Web servers in device engineering (HTTP) or for updating field devices via file upload (FTP). Compatibility is also important for supporting standard synchronization protocols such as IEEE 1588 [10].

2.1.5 Configurability and Extendibility

The standards and devices in the Industrial domain are more difficult to change over time compared to home and office networks. Continuous backward compatibility is required when new devices are added to the existing system. Hence, once a protocol is deployed, it is expected to run for a number of years. Consequently, a RT industrial Ethernet protocol should allow for adding new devices and new applications.

2.1.6 Dynamic Allocation of RT bandwidth

As a recent development, the literature considers that the communication requirements of CIM can dynamically change, depending on the operating condition of the application [12]. For example, the paradigm of self-triggered control [17] allows pre-computing time instants for device level controllers when communication is required. It is also observed that only distributed controllers of currently active system components on the system level need to exchange information for their coordination [18] [19]. Hence, bandwidth only needs to be allocated to currently active system components. Consequently, it is desired that industrial Ethernet protocols allow for dynamically adapting the RT bandwidth allocation depending on the instantaneous bandwidth requirements.

2.2 Real-time Ethernet Implementations

The main motivation of approaches for providing RT support on Ethernet is avoiding or significantly reducing collisions. Existing approaches can be grouped into four main classes [20] [6]. The first class is *the modification of the standard Ethernet interface hardware and software* which includes protocols such as SERCOS III [21] [22], ProfiNet [23] and EtherCAT [24] [25]. While these protocols achieve high performance for a number of the metrics above they do not satisfy the compatibility requirement which was the main drive behind employing Ethernet in the industrial networking rather than proprietary protocols in the first place. The second class of protocols which include MOD-BUS/TCP [26] *operate in the limits of the TCP/IP stack* over standard Ethernet to achieve compatibility. Traffic shaping to achieve low delays on Ethernet is suggested [27]. However, such protocols introduce nondeterministic delays in the communication. The third class is *switched Ethernet* which abandons the shared medium altogether and the fourth class is *adding a new software layer on top of standard Ethernet* which controls the transmission of the traffic on the shared medium. Both of these approaches aim for collision avoidance to archive RT guarantees with standard Ethernet hardware.

2.2.1 Switched Ethernet

The IEEE 802.3x standard replaces shared medium communication by full-duplex switched Ethernet with flow control. This configuration results in the elimination of collisions and [8] argue by means of an example that the response of control applications does not significantly deviate from point to point communication under the delays of (high-speed) switched Ethernet.

Ethernet/IP [28] [29] is realized on the standard TCP/UDP/IP protocol suite over full-duplex switched Ethernet. Ethernet/IP implements the Common Industrial Protocol (CIP) at its upper layers. This protocol makes use of the virtual bridged LAN (VLAN) tags for the prioritization of Ethernet frames with RT data over other frames. However, the queuing of the frames in the switches still introduces non-deterministic delays. The operation correctness heavily relies on the accuracy of the device synchronization.

A number of issues must be considered regarding Switched Ethernet. First of all; although collisions are eliminated, the medium access contention problem is now transferred to the queues in switches [6] [1]. Resolving this contention according to the RT constraints requires scheduling and prioritization capabilities in the network devices [30] [31] as defined in the Ethernet standards IEEE 802.1p and IEEE 802.1Q. Despite scheduling, it is only possible to derive conservative worst-case RT bounds [32] [33]. Furthermore, the precision of synchronization protocols that enable the timing of the communication decrease when switches are introduced in the network [6].

Switched Ethernet implies a star topology which connects multiple devices to a central switch. Over the course of development of industrial networks, the proprietary fieldbus standards were designed for bus or ring topologies to reduce the cabling cost. Hence, implementing a switched Ethernet network requires an individual cable connecting each sensor/actuator/controller to a switch leading to a full re-cabling of the industrial plant [1] [34]. Tree topology instead of star topology is an alternative. However, tree topology results in cascaded switches between end nodes and the queuing delay becomes more prominent [35]. It is interesting to note that the reason for the hardware modifications of a number of Industrial Ethernet protocols is implementing a daisy chain to connect the devices over a bus or a ring [1].

The focus of our work is standard Ethernet based protocols that are designed to operate on shared medium Ethernet without requiring any switches because of the listed limitations of switched Ethernet and requirements for RT guarantees and compatibility. Next, we present a number of protocols that are designed to operate under these conditions.

2.2.2 Shared Medium (IEEE 802.3) with an Additional Medium Access Layer

A number of protocols are proposed to avoid collisions and the subsequent random frame retransmissions of Ethernet's CSMA/CD by implementing an additional layer on top of Ethernet. This layer forwards the messages from RT and nRT applications to the Ethernet MAC layer in a controlled way such that collisions are avoided and certain RT delivery guarantees are achieved. The literature provides different controlled medium access techniques for the realization of this additional layer such as Time Division Multiple Access (TDMA), master-slave operation or token passing [20] [6].

Ethernet for Plant Automation (EPA) [36] [37] employs TDMA which provides deterministic access to the medium via allocating time slots to the nodes by the added layer that is called Communication Schedule Management Entity (CSME). However, only the unique owner of each given time slot can transmit in that slot. Hence, the allocated bandwidth is wasted if a node does not need to transmit during its slot time or does not transmit at all. Such bandwidth waste becomes more significant if the time slot duration is long. Furthermore, redundant slot allocations are necessary for possible message retransmissions in case of error.

On demand guaranteed message transmission instead of static allocation can be achieved with master-slave communications. In this access method a selected master node polls the slave nodes for possible message transmission. The polling time decreases the efficiency of the access and this time is wasted if a polled slave node does not have any frame to transmit. Hence, master-slave organization is appropriate particularly for small numbers of nodes with regular traffic patterns. Master-slave communication comes with several disadvantages. Master-slave access is a centralized architecture requiring relatively high processing capabilities in the master node and making it a single point of failure. Moreover, if the processing capabilities of slave nodes are low, the communication might be slowed down although the available bandwidth is high. Nonetheless, the literature proposes different master-slave industrial Ethernet protocols. Ethernet Powerlink (EPL) [38] [35]

[39] implements an additional layer over standard Ethernet that provides master-slave medium access. It has a computed efficiency of 25% [6], whereby it is observed that the nRT traffic on EPL experiences long latencies.

FTT-Ethernet [9] is a further master-slave protocol which runs with a repeating fixed size elementary cycle (EC). Each EC starts with a trigger message that is broadcast by the master node. This message both serves for synchronization of the nodes and announces the transmission schedule for the periodic messages from all slave nodes. Following the periodic message transmissions, the master node polls the slaves for possible event-triggered transmissions. The implementation of FTT-Ethernet is carried out on the SHaRK real-time kernel. Experimental results show 36% and 11% utilization for time-triggered and event-triggered traffic respectively. Although FTT-Ethernet enables dynamic bandwidth allocation by changing the transmission schedule in different ECs, the master node is required to have full information about the communication requirements of the slave nodes.

Token passing is another method for providing on demand access to nodes on a shared medium. A token is a special control message that represents the right to send a frame on the network. During the operation of the protocol, only a single node holds the token at any time. The token is forwarded to the next node after the frame transmission or a constant idle time if the token holding node does not have any frame to send. One important metric that defines the performance of the token-based protocols is the token rotation time.

A large number of nodes and hence a long token rotation time slows down the operation of the protocol. Furthermore, losing the token message results in the disruption of the operation. Time Critical Control Network (TCNet) [40] is a token-passing based protocol that is implemented on standard Ethernet. The nRT traffic is supported with low priority transmission and synchronization is achieved via the special message carried in the token message. The Virtual Token Passing (VTPE) Protocol [34] prioritizes RT traffic over nRT traffic by disabling the binary exponential back off (BEB) algorithm of Ethernet and setting the back off interval of RT traffic producer nodes to 0. The arbitration among stations that transmit RT traffic is carried out with token-passing. The VTPE protocol is implemented by using Ethernet Controllers that support disabling of the BEB. It is stated that VTPE protocol is able to support the transfer of RT traffic in fully loaded Ethernet networks, where the saturation level for the shared Ethernet networks is above 40% when dealing with small sized messages. The medium access delay is reported to be 5.77ms.

2.2.3 Synchronization

The most significant objective of industrial communication networks is guaranteeing bounded response times for temporally constrained traffic which requires temporal consistency among the participating nodes. The most widely accepted time synchronization protocol for Ethernet-based Industrial Communication Protocols is IEEE 1588 [10] [41] which has a distributed implementation that can seamlessly run on Ethernet. IEEE 1588 operates according to the precision-time protocol (PTP), where a number of message exchanges take place between a selected master node and slave nodes periodically. The slave nodes compute their clock offsets from the master node according to these messages and synchronize their clocks with the master node. IEEE 1588 is adopted as the synchronization protocol by EPA and EPL while industrial communication protocols such as EtherCAT [24] and SERCOS III (IEC 61491) [21] include their own synchronization mechanisms to meet the high precision synchronization requirements.

2.3 Comparison and Discussion

Table 1 summarizes the properties and performance metrics of the described protocols. EPA, EPL and TCNet are industrial standards and each has two different profile definitions. We provide the different metric values for both of these profiles for each protocol in the Table 1. In addition, FTT-Ethernet and VTPE are included in this table. Note that the properties for our protocol D²RIP are obtained in the scope of this thesis.

The meanings of the parameters in Table 1 are:

A/I: Academic/Industrial proposal
DD: Delivery time
B_{eff}: The effective RT throughput
B_{nRT}: Maximal available nRT bandwidth
Sync: Synchronization method

Table 1 Comparison of the Industrial Ethernet Protocols

Protocol	A/I	Medium Access	DD (ms)	Nodes	B_{eff} (Mbit/s)	B_{nRT}	Sync./Accuracy
EPA [36] [1]	I	TDMA/static	5, 0.1	32, 64	12.28	85%	IEEE 1588, 10 µs, 1 µs
EPL [38] [1] [6]	I	Master Slave/static	0.4, 5.5	4, 150	15.2, 32	19.6%, 4.4%	IEEE 1588/1 s
TCNet [40] [1]	I	Token Passing/static	2/20 /200	24, 13	58.4/51.2/7.2, 45.6/40.8/4.8	0%, 20%	Asynchronous/ -
FTT-E [9]	A	Master Slave/dynamic	1	Not specified	36	11%	Synch message/≈ 1ms
VTPE [34]	A	Token Passing/static	5.8	256	≈ 40	Not specified	Asynchronous/ -
D²RIP	**A**	**TDMA/dynamic**	**< 5**	**dynamic**	**16.1**	**> 50%**	**IEEE 1588/1.5 µs**

In view of the requirements stated in Section 2.1, desired properties of industrial Ethernet protocols are

1) Support of RT and nRT traffic
2) Usage of unused RT bandwidth by nRT traffic
3) Dynamic allocation of RT bandwidth depending on communication requirements
4) Synchronization mechanism that complies with industrial standards
5) Distributed implementation on shared-medium Ethernet with COTS hardware

CHAPTER 3

D^{2}RIP PROTOCOL STACK

The D^{2}RIP family as proposed in previous work [12] is a purely distributed protocol that enables communication on shared-medium Ethernet, allows for dynamically changing the bandwidth allocation to different network nodes and supports both RT and nRT communication. That is, this protocol family is particularly useful for control applications that incorporate information about their instantaneous communication requirements. We next give an overview of this protocol family in Section 3.1 and then describe the relevant information about the protocol architecture for our protocol implementation in Section 3.3 to 3.5. Performance parameters are derived in Section 3.6 and Section 3.7 gives an application example.

3.1 Protocol Overview

The protocol stack proposed in [12] is depicted in Figure 1. It is designed to operate on a broadcast network such as *shared-medium Ethernet* and comprises two protocol layers an *Interface Layer* (IL) and a *Coordination Layer* (CL). Here, the IL is used in order to implement time-slotted medium access for both RT and nRT traffic to the broadcast network. The CL performs the essential task of:

i. Deciding for each slot whether it is allocated to RT or nRT traffic.
ii. Determining which node is permitted to transmit a message in case of RT slots.

Hereby, it is foreseen that multiple applications such as distributed RT control applications can be connected to the CL, whereas all messages from nRT applications (such as diagnostics or high-level messages) are directly handled by the IL. As a notable difference to existing protocols, the framework in [12] allows for dynamically changing the slot allocation during run-time based on application specific data. Unique allocation of nRT slots is provided by the IL in this framework. In order to establish a common time base for the time-slotted operation, the framework further includes the use of a synchronization protocol. Together, the proposed framework ensures collision avoidance on the broadcast network by allocating each time slot to a unique node under the assumption that all nodes that operate on a given network segment implement the proposed protocol stack.

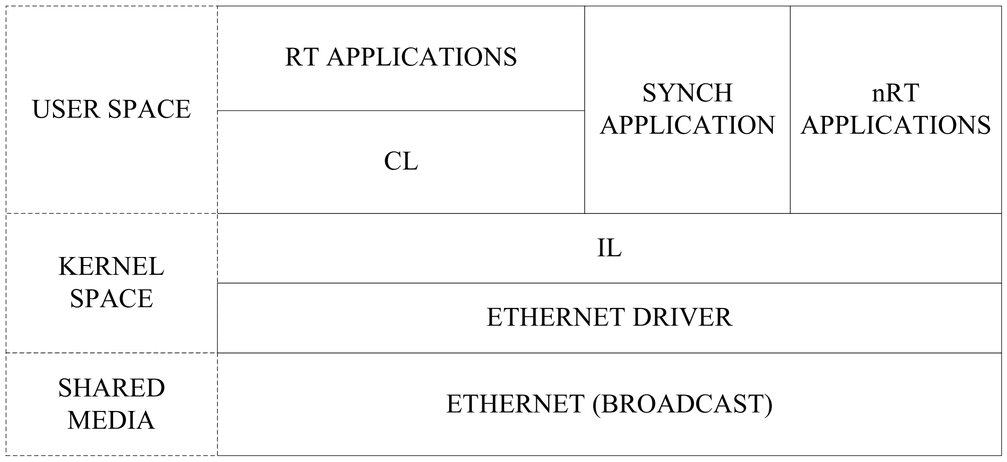


Figure 1 The Architecture of the D²RIP

In the following, we describe the specific properties of the industrial Ethernet protocol D²RIP that is implemented in this thesis. D²RIP constitutes a member of the protocol stack in Figure 1 with specific choices for the CL, IL and the synchronization protocol as is detailed in Section 3.2 to 3.5. Note that a formal description of D²RIP can be found in [12]. The description in this section focuses on the aspects of D²RIP that are relevant for the practical implementation in Chapter 4.

3.2 Distributed Synchronization Protocol

The protocol framework as outlined in Section 3.1 allows the usage of a synchronization protocol outside the defined protocol layers. The task of this module is the precise synchronization of the nodes' clocks which is essential for the time-slotted medium access. D²RIP uses IEEE 1588 [10] as the synchronization protocol, whereby all messages that are sent according to IEEE 1588 are sent as nRT messages. Hence, IEEE 1588 constitutes an independent protocol as required by the protocol framework and could later be replaced by another synchronization mechanism if necessary.

3.3 Coordination Layer Data Structure

Considering that D²RIP is designed to be entirely distributed, the CL operation of each individual network node is identical. Hence, we consider the CL of a generic node *i*, denoted as CL_i, in the sequel. The task of CL_i is to forward RT messages from the application layer (AL) to the IL and vice versa, decide about the type of each time slot (RT or nRT) and uniquely determine the ownership of each RT time slot. To this end, the CL_i is equipped with the following data structures and parameters:

- RT message transmit buffer Tx_i: it is considered that messages can be obtained from different RT applications. Hence, Tx_i is realized as a vector of message buffers, whereby each buffer can hold the current message for one RT application, denoted as *channel*.

- Communication requests req: each req is realized as a four tuple (b, c, eT, dT), whereby b, c, eT and dT are integer values. (b, c, eT , dT) states that node b potentially has an available message for channel c after eT time slots. Moreover, if present, such message must be transmitted within dT time slots. That is, considering a time slot duration of *dSlot*, if a request is issued at time t, it becomes eligible (which indicates that it is valid for processing) at time $ET = t + eT \times dSlot$. Moreover, such request expire at time $DT = t + dT \times dSlot$ (which means that the deadline for message transmission is violated). Hence eT and dT are relative values where ET and DT are *absolute* times.

It has to be noted that the *eligibility time* (eT) parameter is particularly important in practical applications. For example, if programmable logic controllers (PLC) with cyclic operation are used, new data will only be available after the completion of each cycle. Hence, a communication request for such a controller should not be eligible before cycle completion [18] [19]. Accordingly eT and dT are parameters that are specific to each upper layer control application.

- Messages m: each such message consists of data m.data and a set of *communication requests* m.reqs. While m.data contains control application specific information and is hence transparent to the protocol operation, m.reqs contains the communication requests that are relevant for the protocol operation. It is assumed that these requests are provided by each RT application according to the respective communication requirements. It is important to note that requests that are generated by an RT application on node i can allocate RT bandwidth for nodes that are different from i (see also the application example in Section 3.7).

- Priority queue PQ_i: it holds requests in the form of tuples (b, c, ET, DT) as described before. Requests are inserted into PQ_i upon message reception. To this end, PQ_i is sorted such that PQ_i.Top always holds the request (b, c, ET, DT) with the smallest deadline that is at the same time eligible. In each time slot, the highest-priority request is removed from PQ_i and the unique node b that belongs to this request is allowed to send a message for channel c in this time slot.

- Decision variables $RTCL_i$, $myCL_i$, $myCH_i$: in each time slot, $RTCL_i$ is true if there is an eligible request (b, c, ET, DT) in PQ_i.Top. Moreover, $myCL_i$ is true if $myCH_i = c$ and $b = i$, that is, the slot is reserved for channel c of node i. In that case, $myCL_j$ is false and $myCH_j = 0$ for all nodes $j \neq i$. If $RTCL_i$ is false, the current slot is an nRT slot.

Using the data structures and parameters as introduced above, CL_i performs the following actions.

- Messages from RT applications AP2CL: available messages *m* are obtained from the RT applications that are connected to CL_i. Each message is placed in the correct buffer of Tx_i.

- Decision variable update CLUPDATE: the values of the decision variables $RTCL_i$, $myCL_i$ and $myCH_i$ are determined for the current time slot. To this end, PQ_i.Top is checked. If PQ_i.Top = (*b*, *c*, *ET*, *DT*) exists, $RTCL_i$ is set true and $myCL_i$ is set true if $myCH_i$ = *c* and *b* = *i*. Otherwise, $myCL_i$ is false and $myCH_i$ = 0. If PQ_i.Top is empty, $RTCL_i$ is false, $myCL_i$ is false and $myCH_i$ = 0.

- RT message passing to IL CL2ILRT: If $RTCL_i$ is true, $myCL_i$ is true and $myCH_i$ = *c* in the current time slot, node *i* is allowed to transmit a message for channel *c*. If the buffer Tx_i for channel *c* is empty, no message was provided by the application, yet. In that case, *m*.req = (*i*, *c*, *eT*, *dT*) is set in order to request a further time slot for transmission of the application message for channel *c* of node *i*. The data field of the message *m*.data remains empty. Otherwise, the application message in the buffer Tx_i is forwarded to IL. If $RTCL_i$ is false, it is indicated to IL that the current slot is a nRT slot without passing any message.

- RT message reception from IL IL2CLRT: If an RT message is received in the current time slot from IL, PQ_i.Top is removed from PQ_i.

- RT message forwarding to the RT applications CL2AP: If the message data *m*.data received from the IL is not empty, then *m* is a valid application message from some node. In that case, the message data is forwarded to the RT applications.

The message transmission is depicted in Figure 2. We again emphasize that the same computations are performed by the CL of each node. As a result, it is ensured that the evaluation of each CL parameter is the same for each node in each time slot [12]. In particular, each node maintains the same priority queue and hence determines consistent values for the decision variables $RTCL_i$, $myCL_i$ and $myCH_i$: whenever $RTCL_i$ is true for some node *i* in a certain time slot, it is true for all other nodes in the same time slot. In addition, if $myCL_i$ is true for node *i*, it is false for all other nodes. Hence, always one unique node is allowed to transmit per RT slot and this information is available to all nodes. Likewise, all nodes know if a time slot is designated as nRT slot.

3.4 Interface Layer Data Structure

In analogy to the CL, the IL operation of each individual network node is identical. Hence, we consider the IL of a generic node *i*, denoted as IL_i, in the sequel. The task of IL_i is to provide time-slotted access to the underlying broadcast network depending on the information provided by the connected CL_i. That is, if CL_i decides that the current time slot is designated for nRT traffic, then the ILs of all nodes apply a consistent transmission schedule in order to determine a unique sender node. In case of an RT slot, the IL forwards the message from the CL of the unique node that owns the time slot to the broadcast network. The IL operation is realized by help of the following data structures and parameters.

- RT message transmission buffer $TxRT_i$: it stores the current RT message to be transmitted to the broadcast network.

- nRT message transmission queue $TxnRT_i$: it is realized as a FIFO queue that contains messages from nRT applications that are to be transmitted in available nRT slots. $TxnRT_i$.Top shows the first message in the queue.

- Received nRT message $RxnRT_i$: it stores the nRT message that is received from the broadcast network.

- Received RT message $RxRT_i$: it stores the RT message that is received from the broadcast network.

- RT decision variables $RTIL_i$ and $myIL_i$: if the current slot is an RT slot, $RTIL_i$ is true. If additionally, the slot is allocated to node *i* $myIL_i$ is true. Otherwise $RTIL_i$ and $myIL_i$ are false.

- nRT decision variables cnt_i, cyc_i and $nRTSet_i$: IL_i maintains a counter cnt_i that is incremented in each nRT slot and that is reset to 0 after cyc_i nRT time slots. That is, cnt_i counts the number of nRT slots modulo cyc_i. The variable $nRTSet_i$ is realized as a set, whereby each member of the set represents the number of an nRT slot where node *i* is allowed to transmit. In order to make sure that each nRT slot is allocated to a unique node, it is required that $nRTSet_i \cap nRTSet_j = \emptyset$ for any different nodes *i* and *j*.

Using the data structures and parameters as defined above, the IL performs the following actions.

- RT message passing from CL CL2ILRT: the decision variables $RTIL_i$ and $myIL_i$ are set to the respective values $RTCL_i$ and $myCL_i$ of the CL. If $RTCL_i$ and $myCL_i$ are both true, an RT message *m* is passed from the CL and is stored in the buffer $TxRT_i$.

- Message passing to the shared medium IL2SM: if $RTIL_i$ and $myIL_i$ are both true, the RT message in $TxRT_i$ is transmitted to the shared medium. Otherwise, if $RTIL_i$ is false, it is checked if the current count value cnt_i belongs to $nRTSet_i$. In the positive case, the first nRT message in the FIFO queue $TxnRT_i$ is transmitted to the broadcast network.

- Message reception from the shared medium SM2IL: if a message *m* is received from the broadcast network, $RTIL_i$ is checked. If $RTIL_i$ is true (RT message), m is stored in $RxRT_i$. Otherwise, *m* is stored in $RxnRT_i$ and the counter cnt_i is incremented.

- RT message forwarding to the CL IL2CLRT: if $RTIL_i$ is true (RT slot) and $RxRT_i$ is not empty (RT message received), the message is forwarded to the CL.

- nRT message passing from nRT applications AP2ILNRT: the application layer can pass nRT messages to the IL. Each such message is stored in the FIFO queue $TxnRT_i$. Note that as shown in Figure 2, such nRT messages directly go to IL without any interface to CL.

- nRT message forwarding to nRT applications IL2APNRT: IL initiates the forwarding of the nRT message to the respective applications from the $RxnRT_i$ buffer.

The message transmission is depicted in Figure 2. In analogy to the CL, the computations performed by the IL of each node are identical. Considering that consistent data are provided by the CL, this ensures that all IL parameters are consistent in each time slot [12] and a unique node is identified for transmission of both RT and nRT messages. Moreover, considering that all messages are transmitted on a broadcast network, the IL of each node receives the same message in each time slot.

3.5 Sequential Protocol Operation in each Time Slot

In order to clarify the protocol operation, we next point out the sequential actions that are taken by the different layers in each time slot as depicted in Figure 3 and Figure 4, whereby (a) represents the case of a node that owns the current time slot and (b) represents the case of a node that does not own the current time slot. Figure 3 shows the RT message transmission case and Figure 4 shows the nRT message transmission case.

We first consider with the assumption that the considered node is supposed to transmit an RT or nRT message. At the beginning of the slot, the current messages of connected RT applications are polled (if available). Next, the action CLUPDATE is performed in order to determine the type (RT or nRT) and the owner of the slot. In case of an RT message that is transmitted by the specified node (Figure 3a), the message is forwarded to the IL with the action CL2ILRT including the slot ownership information. In case of an nRT message that is transmitted by the node (Figure 4a), CL2ILRT indicates that the current slot is an nRT slot. That is, the ownership of the slot by the node is determined by looking at cnt_i and $nRTSet_i$. After that, IL2SM transmits the RT or nRT message on the broadcast network. After the reception of the message by all nodes including the sender node the message is passed to the IL and either forwarded to the nRT application with IL2APNRT (nRT slot) or to the CL with IL2CLRT (RT slot). All relevant parameters such as PQ_i are updated and the message is forwarded to the connected RT applications with CL2APRT. Finally, note that the nRT application is permitted to add an nRT message to the FIFO queue $TxnRT_i$ at any time.

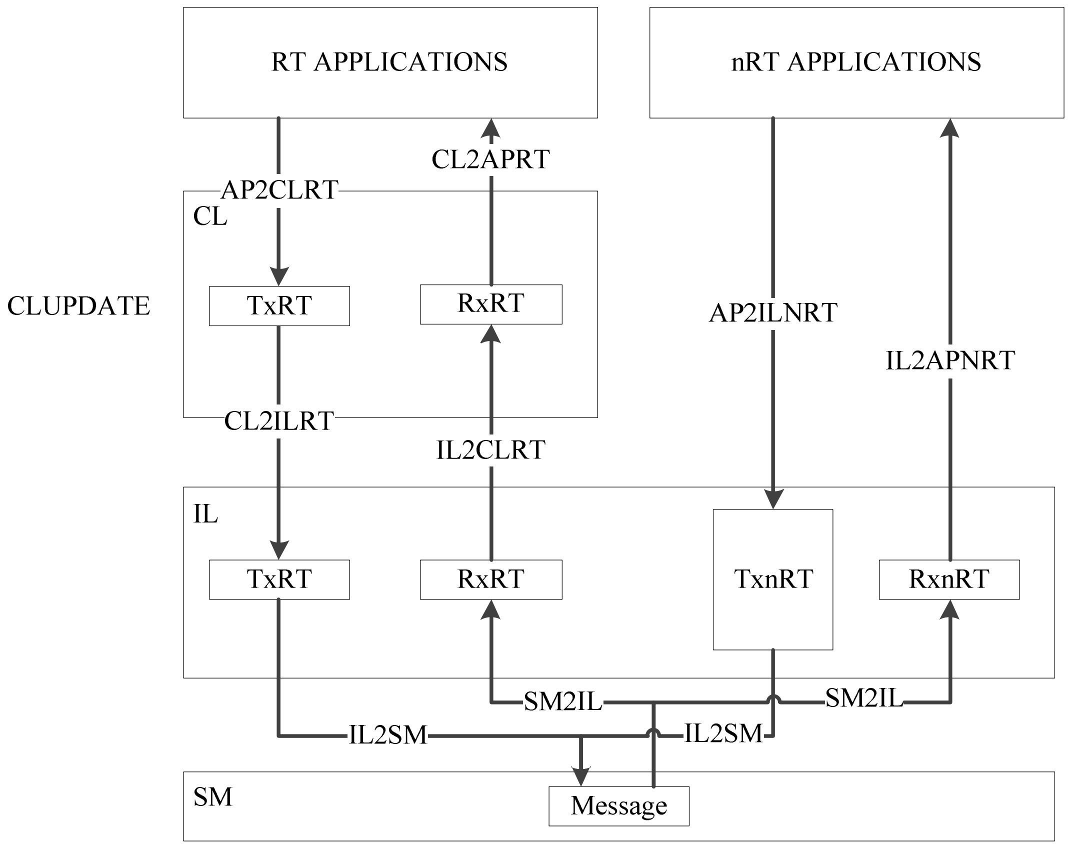


Figure 2 Message Transmission of the D²RIP

Second, we describe Figure 3b and Figure 4b with the assumption that the considered node does not own the current time slot. Then, different from the previous case, CLUPDATE either finds that the current slot is an nRT slot or an RT slot that is not owned by the considered node ($myCL_i$ is false). This information is transferred to the IL via CL2ILRT. If the current slot is an RT slot, $myIL_i$ is false and the node does not transmit a message. If the current slot is an nRT slot, it holds that cnt_i does not belong to $nRTSet_i$. Again, no message is transmitted. However, the considered node will receive an nRT or RT message from another node that owns the current time slot with SM2IL. After this reception, the same actions as in the first case are taken.

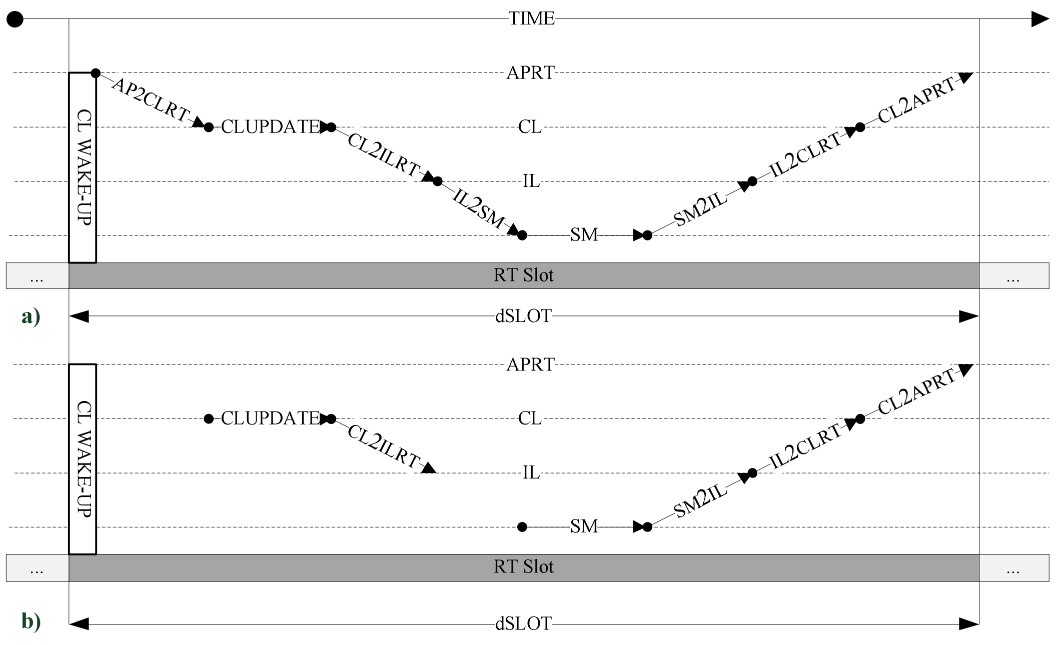


Figure 3 Sequential Actions for the RT Message Transmission in a Time Slot

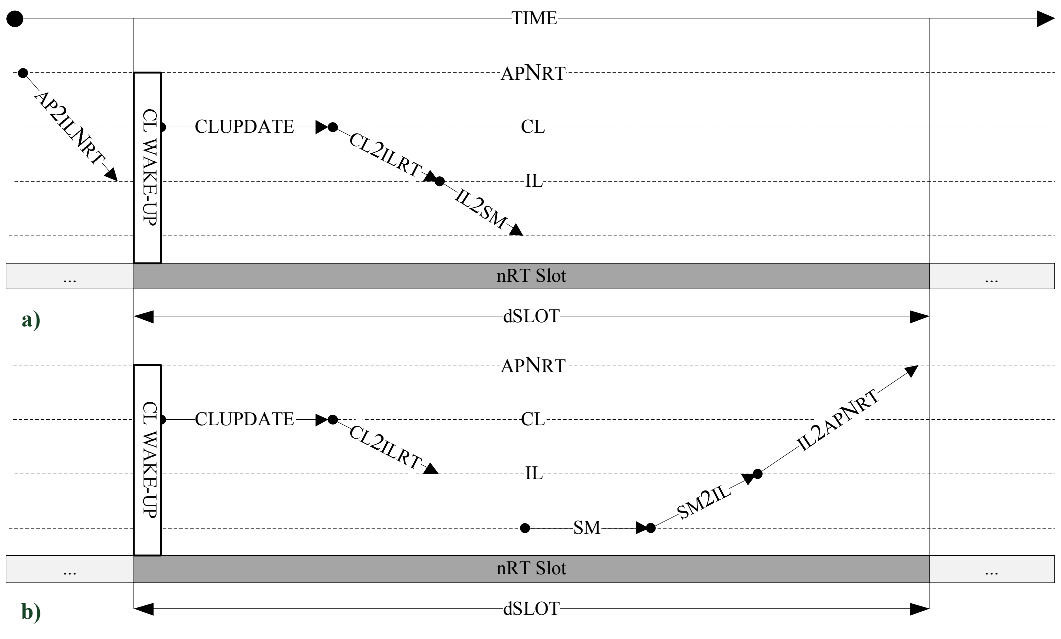


Figure 4 Sequential Actions for the nRT Message Transmission in a Time Slot

We finally note that the same actions are performed in each time slot, whereby always at most one node has the right to transmit and all other nodes listen to and process the messages transmitted on the broadcast network. As an important feature of the proposed protocol, all RT messages are received by the CL of each node such that the CL data are always consistent among all nodes. Moreover, the operation of the IL of each node maintains consistent information about the ownership of nRT slots.

3.6 Performance Parameters of D²RIP

Based on the protocol operation as described in the previous sections, we now evaluate several performance metrics as discussed in Section 2.1. To this end, we introduce network specific, protocol specific and application specific parameters as summarized in Table 2.

Table 2 Performance Parameters of D²RIP

	Network Specific Parameters
$\boldsymbol{B}$	Bandwidth of the underlying broadcast network
	Protocol Specific Parameters
dSlot	Duration of each time slot
$\boldsymbol{L}$	Length of an RT message in bit
	Application Specific Parameters
$\boldsymbol{B_{sum}}$	Sum of all required RT bandwidth allocations for all nodes (static allocation)
$\boldsymbol{B_{max}}$	Maximum required RT bandwidth allocation
$\boldsymbol{B_{min}}$	Minimum required RT bandwidth allocation
$\boldsymbol{B_{eff}}$	Effective total bandwidth
$\boldsymbol{eT_{req}}$	Eligibility time of request *req*
$\boldsymbol{dT_{req}}$	Deadline time of request *req*
$\boldsymbol{Q_{req}}$	Maximum number of requests in the priority queue together with request *req*

B_{sum} represents the sum of all worst-case RT bandwidths that are required in all network nodes. This bandwidth has to be allocated in industrial communication protocols that rely on *static* medium access. In addition, Table 2 introduces the parameters B_{max} and B_{min} that are specific to D²RIP. They capture the fact that, it is usually not the case that all nodes have to communicate with their worst-case bandwidth allocation at any time. That is, B_{max} represents the maximum amount of allocated RT bandwidth that can be required at any time instant. On the other hand, B_{min} is the minimum amount of allocated RT bandwidth at any time instant. Hence, the RT bandwidth allocation for D²RIP dynamically changes between B_{min} and B_{max} during system operation, whereby B_{max} is usually considerably smaller than B_{sum}. In addition, it can be assumed that the eligibility times and deadlines of all requests are known from the application design. Hence, their maximum and minimum values can be easily determined. Finally, the application analysis usually allows computing a bound Q_{req} on the number of requests that can be in front of a request req in the priority queue. An example for such evaluation is given in Section 3.7.

It is now possible to evaluate the performance metrics in analogy to Table 2. Note that numerical values for the different performance metrics are given in the experimental evaluation in Chapter 5. The effective bandwidth B_{eff} of D^2RIP is given by

$$B_{eff} = \frac{L}{dSlot} \tag{3.1}$$

Furthermore, the previous work [19] establishes a relation between the application parameters and the time slot size. It must hold that

$$dSlot \leq \frac{dT_{req} - eT_{req}}{Q_{req} + 1} \tag{3.2}$$

In turn, the worst-case delivery time w_{req} associated to request req can be computed as

$$W_{req} = dSlot \times \left(Q_{req} + 1\right) + eT_{req} \tag{3.3}$$

and it can be concluded that the minimum deadline for any request req that is supported by D^2RIP is given by

$$dT_{req} \geq w_{req} \tag{3.4}$$

Considering (3.1) and (3.4) it is clearly desired to make *dSlot* as small as possible in the practical protocol implementation. We further evaluate the advantage of D^2RIP in comparison with industrial Ethernet protocols with static bandwidth allocation. The RT bandwidth gain G_{RT} can be quantified by the ratio

$$G_{RT} = \frac{B_{sum}}{B_{max}} \tag{3.5}$$

We further consider that *n* network nodes are transmitting RT messages, whereas the worst-case RT bandwidth requirement per node is given by $B_{wc,i}$ for $i = 1,..,n$. Then, the overall required RT bandwidth in case of a static slot allocation is

$$B_{sum} = \sum_{i=1}^{n} B_{wc,i} \tag{3.6}$$

Hence, for a static bandwidth allocation, $B_{sum} \leq B_{eff}$ is required even if the required RT bandwidth changes instantaneously. This is different for protocols such as D^2RIP that support dynamic bandwidth allocation. In addition, the available nRT bandwidth B_{nRT} changes between

$$B_{eff} - B_{max} \leq B_{nRT} \leq B_{eff} - B_{min} \tag{3.7}$$

From this equation, it can be seen that all the effective bandwidth could be allocated to nRT traffic if desired.

3.7 Application Example

We illustrate the operation of D²RIP by an application example that is adapted from [12]. We consider the workcell in Figure 5 that comprises 3 components – a robot (R) that can move a robot arm, a conveyor (C) that can transport products in two directions, and a painting device (PD), that paints parts using a machine spray gun. We assume that each component is controlled by an individual programmable logic controller (PLC), and there is one cell PLC that coordinates the operation of the different components. We denote the respective PLCs as PLC-R, PLC-C, PLC-PD and PLC-S.

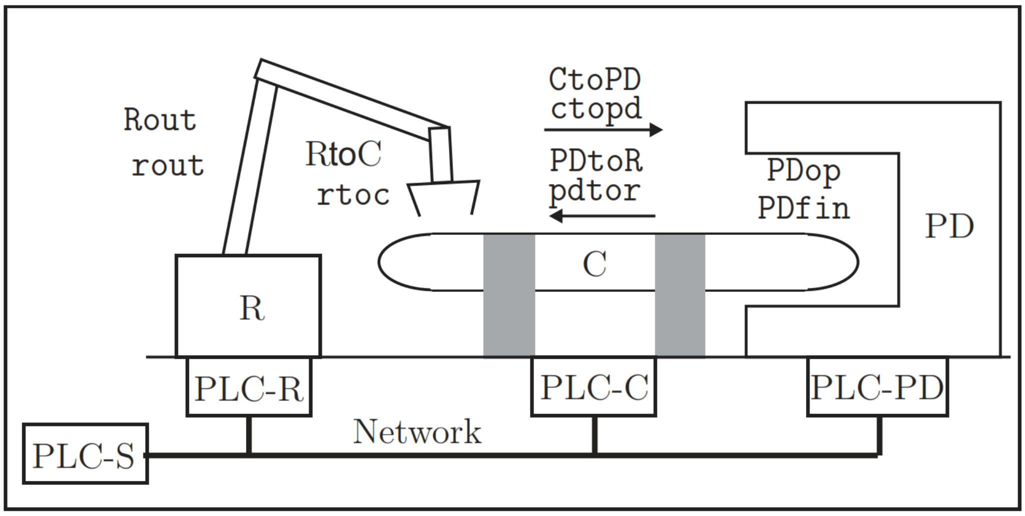


Figure 5 Workcell: Robot, Conveyor and a Painting Device

The desired operation of the workcell is to move products to PD starting from R and via C. Products at PD are painted and leave the workcell from R after completion. Communication among the PLCs is required in order to coordinate the operation of the different system components. We use event-based communication between the coordinator PLC-S and the component PLCs. The events and their related PLCs are listed in Table 3 and an example of event-based communication is illustrated in Figure 6. For example, RtoC/rtoC represent the start/completion of a product transport from R to C or CtoPD/ctopd indicate the start/completion of product transport from C to PD. The occurrence of RtoC/rtoC requires communication between PLC-S and PLC-R, whereas the occurrence of CtoPD/ctopd is communicated between PLC-S and PLC-C. Note that each operation is represented by a “start” and a “completion” event. The start event indicates start of operation of the related component PLC that performs local control until the operation is finished. That is, the finish event is sporadically generated by the component PLC.

In analogy to previous work [18] [19] [12], we assume a three-step communication for each event that is carried out as follows:

1) The coordinator PLC-S *queries* the related component PLC to check if it is able to execute the event. Such query is indicated by a question mark in front of the respective event.
2) The queried component PLC responds with a *notifications* to PLC-S whenever it is ready to execute the event. A notification is indicated by an exclamation mark in front of the respective event.
3) PLC-S receives all notifications for the event it issues a single *command* to all PLCs that are related to the event to execute the event synchronously.

Table 3 Actions of the Workcell Example

Event	Description	Related PLCs
RtoC	initiate product transport from R to C	PLC-R, PLC-S
rtoc	complete product transport from R to C	PLC-R, PLC-S
CtoPD	initiate product transport from C to PD	PLC-C, PLC-S
ctopd	complete product transport from C to PD	PLC-C, PLC-S
PDop	PD starts operation	PLC-PD, PLC-S
PDfin	PD finishes operation	PLC-PD, PLC-S
PDtoR	initiate transport from PD to R	PLC-C, PLC-S
pdtor	complete transport from PD to R	PLC-C, PLC-S
Rout	initiate product output from R	PLC-R, PLC-S
rout	complete product output from R	PLC-R, PLC-S

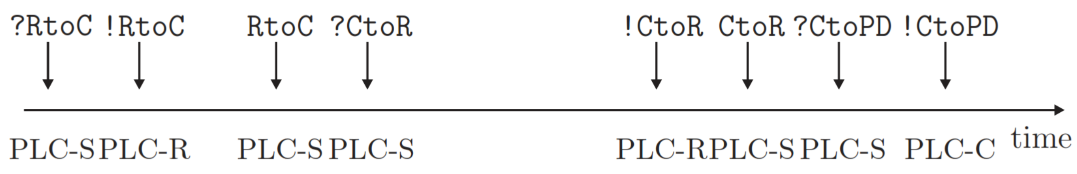


Figure 6 Timing Diagram for the PLC Communication of the Example Workcell

It can be seen from the workcell example that, although there are four nodes on the network, only one of them is expected to transmit at any time instant. For example, initially, PLC-S will query PLC-R for possible product transport, whereby all other PLCs (PLC-R, PLC-C and PLC-PD) are not required to communicate. After that, only PLC-R is supposed to respond to the query and again only PLC-S is supposed to send a command afterwards. Hence, it is possible to dynamically allocate bandwidth only to the PLC that is expected to send a message. Note that such task cannot be accomplished using the existing industrial Ethernet protocols as described in Section 2.2 except for the master-slave protocol FTT-Ethernet. We next explain how our proposed protocol D²RIP can be used to realize the communication for the workcell example. To this end, we list the request to be transmitted together with the respective questions, notifications and commands of the different PLCs in Table 4. Here, we consider that only one RT application is connected to D²RIP on each PLC. Hence, there is only one channel for each device. We further assume that the deadline for each message transmission is 5 ms, whereas the eligibility time is given by 4 ms.

Table 4 Communication Requests for the Workcell Example

Question	**Request**	**Notification**	**Request**	**Command**	**Request**
?RtoC	(PLC-S,1,4,5)	**!RtoC**	(PLC-R,1,4,5)	**RtoC**	(PLS-S,1,4,5)
?rtoc	(PLC-S,1,4,5)	**!rtoc**	(PLC-R,1,4,5)	**rtoc**	(PLC-S,1,4,5)
?CtoPD	(PLC-S,1,4,5)	**!CtoPD**	(PLC-C,1,4,5)	**CtoPD**	(PLC-S,1,4,5)
?ctopd	(PLC-S,1,4,5)	**!ctopd**	(PLC-C,1,4,5)	**ctopd**	(PLC-S,1,4,5)
?PDop	(PLC-S,1,4,5)	**!PDop**	(PLC-PD,1,4,5)	**PDop**	(PLC-S,1,4,5)
?PDfin	(PLC-S,1,4,5)	**!PDfin**	(PLC-PD,1,4,5)	**PDfin**	(PLC-S,1,4,5)
?PDtoR	(PLC-S,1,4,5)	**!PDtoR**	(PLC-C,1,4,5)	**PDtoR**	(PLC-S,1,4,5)
?pdtor	(PLC-S,1,4,5)	**!pdtor**	(PLC-C,1,4,5)	**pdtor**	(PLC-S,1,4,5)
?Rout	(PLC-S,1,4,5)	**!Rout**	(PLC-R,1,4,5)	**Rout**	(PLC-S,1,4,5)
?rout	(PLC-S,1,4,5)	**!rout**	(PLC-R,1,4,5)	**rout**	(PLC-S,1,4,5)

It is now possible to evaluate the application specific performance metrics as introduced in Section 3.6 for the workcell example. It is readily observed that at most one communication request is in the priority queue at any time instant since only one PLC is supposed to transmit at any time and each PLC only has one channel. Hence, $Q_{max} = 1$. Considering that each deadline is $dT = 5$ ms and each eligibility time is $eT = 4$ ms, it is possible to evaluate the maximum time slot size and overall worst-case delivery time w_{max} according to (3.2) and (3.3):

$$dSlot \ \leq \ \frac{5\ ms - 4\ ms}{2} = 0.5ms \tag{3.8}$$

$$w_{max} = 2\ \times dSlot + 4\ ms \tag{3.9}$$

Furthermore, it is possible to compute the values B_{sum} and B_{max} for the workcell example. In order to meet all deadlines, a static allocation requires reserving one time slot per PLC every 5 ms. We have 4 controllers in our example. Hence, we obtain

$$B_{sum} = 4 \times \frac{dSlot}{5\ ms} \times\ B_{eff} \tag{3.10}$$

According to the operation of D²RIP, the maximum amount of bandwidth for the RT messages is obtained if a message is sent immediately when its associated request becomes eligible. Hence,

$$B_{max} = \frac{dSlot}{4\ ms} \times B_{eff} \tag{3.11}$$

Note that the performance evaluation of the D²RIP implementation in Chapter 5 is based on this workcell example.

CHAPTER 4

D^{2}RIP IMPLEMENTATION

This chapter gives an overview of our D^{2}RIP implementation. Section 4.1 describes the implementation environment and our time synchronization solution is presented in Section 4.2. Section 4.3 explains the control application interface for the D^{2}RIP and Section 4.4 explains our data encapsulation. The protocol details for D^{2}RIP are explained in Section 4.5, 4.6, 4.7 and 4.8. In Section 4.9, we explain D^{2}RIP changes according to the previous implementation. As a general comment, we note that all the functions that implement the D^{2}RIP stack include error checks for unexpected inputs.

4.1 Implementation Environment

Operating system (OS) choice and performance is very crucial for the D^{2}RIP. It should give high performance and reliability when D^{2}RIP runs on it. Thus in this section, we describe how OS is selected, installed, configured and customized.

4.1.1 Real-Time Operating System

A real-time operating system (RTOS) is specially designed to run applications or tasks with high precise timing. This OS is essential in measurement and automation systems where timing is costly or a task delay could cause a system failure. To call an OS as a RTOS, it must have a *known maximum latency* for each of the critical operations that it runs [42]. The D^{2}RIP stack is designed to provide guaranteed bounded delays for RT application data. To this end, it is necessary to implement it on a RTOS which provides the necessary task prioritization mechanisms.

4.1.2 Choosing an Operating System

Linux OS has an advantage over other operating systems for D^{2}RIP because of *open source* [43]. That means that everyone can inspect or alter the OS source code and driver codes as their fit. In D^{2}RIP, Ethernet driver should be modified to collect non real-time packets and to prioritize the real-time packets over the non-real time-packets.

There are currently over six hundred Linux distributions and three hundred of those are in active development, including Debian, Fedora, openSUSE, Arch Linux etc. *Ubuntu* is the most popular and most well-supported distribution available today. *Lubuntu* OS which is based on the Ubuntu release focuses on *speed* and *energy-efficiency* and has *very low hardware requirements* [44]. D²RIP is intended to run on both PCs and embedded controller devices such as Programmable Logic Controllers (PLCs) with small built-in RAM and limited processor capabilities. Hence, we chose Lubuntu as a fast and lightweight OS with very low hardware requirements. The latest version of Lubuntu OS that currently available is *12.10* [44].

4.1.3 Choosing a Disk for Installation

We developed the D²RIP in an entirely portable fashion to be distributed on a single USB flash disk including the RTOS. It should be easily set up in all industrial control systems and start to operate without installing any specific OS. In addition, the D²RIP implementation should be as hardware-independent as possible. There is a wide variety of Programmable Logic Controllers (PLCs) that run different operating system on different hardware. For example, *Wago-I/O-IPC* in Figure 7 that is generally used in the control and it is a target hardware platform to implement D²RIP. It has no built-in hard disk and neither PCI nor PCI-E slots. The only way to boot up OS on *Wago-I/O-IPC* is usb flash disk. On the one hand, flash disks have many advantages over hard-disk drives: *less power consumption*, *no moving parts*, *minimal access time*, *small-size and low cost*, *high degree portability* and *compatibility*.

Figure 7 Wago-I/O-IPC

On the other hand, installing OS on the usb disk brings some drawbacks. However, these drawbacks can be easily solved with the help of a number of OS settings, listed in Table 5.

Table 5 USB-Disk Drawbacks and Solutions

Limitations	Solutions
Limited disk capacity	No need the large disk space to run D^2RIP (only 2 gb disk space will be used)
Limited write speed	Decrease disk writing activities (write data to RAM instead of writing to disk)

4.1.4 Installation of Linux

Before starting the installation process, the appropriate filesystem should be formatted to usb flash disk. *ext4* filesystem is probably the best choice for our Linux OS. Firstly, flash disk master boot record (MBR) section and partition table should be zeroed to avoid problems with different system MBR or previous partitioning.

```
sudo dd if=/dev/zero of=/dev/sdb bs=2048 count=1
```

Next usb flash disk is partitioned by using the *fdisk* command.

```
sudo fdisk /dev/sdb
```

At the *fdisk* prompt, a new partition should be created and this partition should be set as a active and primary partition. We created one partition, which will host the Linux OS. Now usb flash disk is formatted as an *ext4* filesystem with the label Lubuntu with this code:

```
sudo mkfs.ext4 -O ^has_journal -L Lubuntu /dev/sdb1
```

The filesystem of the usb flash disk is ready to install Lubuntu OS. The Lubuntu OS can be installed with the Lubuntu installation CD. There is no need to install Linux OS into usb flash drive every time. Once the OS is installed, then the image of the OS can be taken from the usb flash drive and restored to the others usb flash drives in Linux with these commands:

```
dd if=/dev/sdb of=~/disk.img
dd if=disk.img of=/dev/sdc
```

In Windows OS, there is a tool named “USB Image Tool” that can create images of USB flash drives and restore images of USB flash drives [45].

4.1.5 Configuration and Installation of Real-Time Kernel

Many Linux distributions come with the standard Linux kernel that is not suitable for RTOS because they are designed to perform daily user’s tasks. Therefore, we should patch the Linux kernel with RT-Preempt kernel to obtain RTOS. The latest stable mainline kernel for which the real-time patches are being developed is *3.6.2*, though currently *3.8.2* is available [46]. Before starting the patch-process, first the required software packages should be installed:

```
sudo apt-get install kernel-package fakeroot build-essential libncurses5-dev
```

Then user should get the main 3.6.2 kernel and RT patch, decompress mainline kernel files and patch the mainline kernel with RT patch:

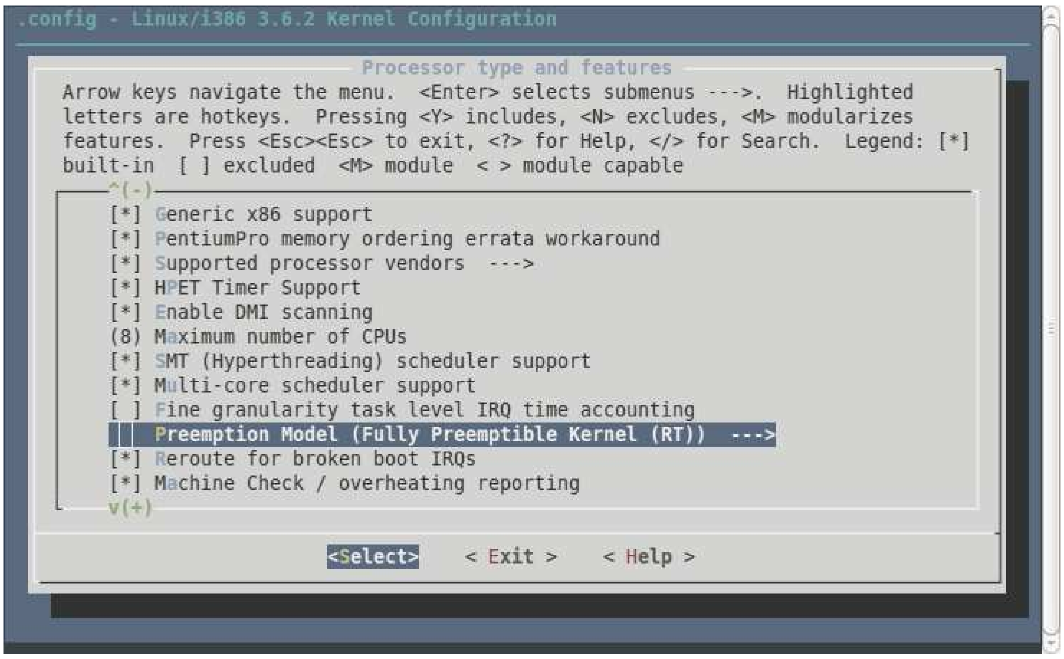
```
mkdir -p ~/tmp/lubuntu-rt
cd ~/tmp/ lubuntu-rt
wget http://www.kernel.org/pub/linux/kernel/v3.x/linux-3.6.2.tar.bz2
wget http://www.kernel.org/pub/linux/kernel/projects/rt/3.6.2/patch-3.6.2-rt4.patch.bz2
tar xjvf linux-3.6.2.tar.bz2
cd linux-3.6.2
patch -p1 < <(bunzip2 -c ../ patch-3.6.2-rt4.patch.bz2)
```

We note that, before building a RT patched kernel, several changes should be made to the kernel configuration file similar to the implementation in [47]. Configuration of the RT kernel settings can be done with the code:

```
make menuconfig
```

The configuration screen of the Linux kernel is shown in Figure 8. The list of the configuration settings is:

- Enable “Tickless System (Dynamics Ticks)”
- Enable “High Resolution Timer Support”
- Set “Preemption Model” to “Fully Preemptible Kernel(RT)”
- Set “Timer frequency” to “1000Hz”
- Disable “Suspend to RAM and standby”
- Enable “Timestamping in PHY devices”
- Enable “PTP Hardware Clock (PHC)”
- Enable “PTP clock support”
- Disable “Show timing information on printks”
- Set ”I/O scheduler” to “Deadline”



```
.config - Linux/i386 3.6.2 Kernel Configuration
                    Processor type and features
  Arrow keys navigate the menu.  <Enter> selects submenus --->.  Highlighted
  letters are hotkeys.  Pressing <Y> includes, <N> excludes, <M> modularizes
  features.  Press <Esc><Esc> to exit, <?> for Help, </> for Search.  Legend: [*]
  built-in  [ ] excluded  <M> module  < > module capable
      ^(-)
      [*] Generic x86 support
      [*] PentiumPro memory ordering errata workaround
      [*] Supported processor vendors  --->
      [*] HPET Timer Support
      [*] Enable DMI scanning
      (8) Maximum number of CPUs
      [*] SMT (Hyperthreading) scheduler support
      [*] Multi-core scheduler support
      [ ] Fine granularity task level IRQ time accounting
          Preemption Model (Fully Preemptible Kernel (RT))  --->
      [*] Reroute for broken boot IRQs
      [*] Machine Check / overheating reporting
      v(+)

                   <Select>    < Exit >    < Help >
```

Figure 8 Linux Kernel Configuration

The RT kernel can be built with this command:

```
sed -rie 's/echo "\+"/#echo "\+"/' scripts/setlocalversion
make-kpkg clean
CONCURRENCY_LEVEL=$(getconf _NPROCESSORS_ONLN) fakeroot make-kpkg -
-initrd --revision=0 kernel_image kernel_headers
```

Finally the RT kernel can be installed to Lubuntu:

```
sudo dpkg -i ../linux-{headers,image}-3.6.2-rt4_0_*.deb
```

4.1.6 Tweaking the Linux Operating System

Linux OS is highly customizable according to D^{2}RIP needs. All settings and configuration files of the OS can be easily edited and saved permanently. Linux, by default, is tweaked for hard drives, which has a negative impact on the performance and longevity of flash-based Linux installs. The flash disk read speed is good enough, but the write speed is not so good. Thus, the purpose of the tweaks is to minimize the amount of unnecessary writes to the usb flash disk. Linux writes to a file's metadata whenever the file is accessed. This is important for some applications such as mail servers, but for D^{2}RIP it will be fine to disable.

In /etc/fstab:
```
UUID=98181808-ec88-4dcd-a94d-bfd79862bd59                    /                    ext4
noatime,discard,data=ordered,errors=remount-ro    0    1
```

Linux can be configured that all logs get written to RAM memory rather than the flash disk. It is not important to keep log files after a system boot.

In “/etc/fstab”:
```
tmpfs /tmp tmpfs defaults,noatime,mode=1777 0 0
tmpfs /var/tmp tmpfs defaults,noatime,mode=1777 0 0
tmpfs /var/log tmpfs defaults,noatime,mode=0755 0 0
tmpfs /var/log/apt tmpfs defaults,noatime 0 0
```

Linux swaps out less used RAM pages into disk when the system needs more memory than is physically available. However, swapping process degrades system performance. Compared to RAM, disks are very slow. Turning off the swappiness will minimize the amount of unnecessary writes to the usb disk.

In /etc/sysctl.conf:
```
vm.swappiness = 0
vm.dirty_writeback_centisecs = 360000
vm.dirty_expire_centisecs = 360000
vm.dirty_background_ratio = 1
```

Also Bash shell logs to the usb disk every command that user type. This situation can be disabled with these settings:

In “/root/.bashrc”
```
export HISTSIZE=0
export HISTFILESIZE=0
unset HISTSIZE
```

Moreover, all unnecessary visual effects are closed to increase overall system performance, such that translucent windows, shadow effects, animations etc. Linux boot-up duration also can be improved by turning off the redundant services with the tool named *sysv-rc-conf*. The remaining services are: Halt, Killprocs, Networking, Reboot, Sendsigs, Single, Umountfs, Umountroot and Urandom.

D^{2}RIP needs to work only with the network interface. Thus many modules that come with Linux installation are unnecessary. These modules send continuously interrupts that degrade the performance of the RT kernel. Therefore, unnecessary devices and device drivers (e.g. sound card, bluetooth, serial port, wifi adaptor ..) were disabled.

Several BIOS settings were modified such as disabling integrated sound card, power saving features (C2 State: Disable CPU C State: Disable, CPU Idle State: High Performance), processor frequency scaling, dynamic fan control and thermal monitoring, event log etc. These functions use dedicated nonmaskable interrupts that cannot be disabled by the OS. When such interrupts are active during real-time operation, they can cause unacceptable jitter and latency. The unused system modules are disabled, floopy, bnep, bluetooth, parport, coretemp etc.

To ensure high performance, *NAPI* (Rx polling mode) property of the network interface card was activated. With *NAPI* configuration, interrupts for receiving are decreased and overall network performance is improved. According to e1000e module documents [48], the Ethernet driver can limit the amount of interrupts per second that the adapter generates for incoming packets. D^{2}RIP needs to have low latency for incoming packets. Therefore, *InterruptThrottleRate* was set to 0 to turn off any interrupt moderation and may improve small packet latency. Another tunable setting for Ethernet module is about the generation of transmit interrupts. Setting up *TxIntDelay* to 0 disables the property and starts the transmission of the packet immediately.

```
insmod e1000e.ko InterruptThrottleRate=0,0 TxIntDelay=0,0
```

Moreover, the Ethernet (IEEE 802.3) cannot provide a real-time response for packet transmission. CSMA/CD algorithm can detect collisions, so it schedules packet retransmission. This property should be disabled to bound packet transmission time. The re-transmission option of the Ethernet is disabled with the setting in the Ethernet driver:

```
E1000_COLLISION_THRESHOLD = 0
```

Linux has the ability to assign certain interrupts (IRQs) to specific processors or groups of processors. This is known as *SMP affinity* (proper interrupt handling), and it allows the user control how the OS will respond to various hardware events. In order to use this property, system should have more than one processor. Our experiments computers have one Intel processor with four cores. Firstly, the Ethernet’s IRQ should be figured out by reading “/proc/interrupts” file to set the packet transmit and receive IRQ.

```
          CPU0      CPU1         CPU2      CPU3
72:      87298         0            0         0    PCI-MSI-edge     eth0-rx-0
73:      93707         0            0         0    PCI-MSI-edge     eth0-tx-0
```

To dedicate four CPU cores for transmit and receive IRQ:

```
echo 00000f > /proc/irq/72/smp_affinity # eth0-rx-0
echo 00000f > /proc/irq/73/smp_affinity # eth0-tx-0
```

4.2 Synchronization with IEEE 1588

One of the main objectives of the industrial communication networks is guaranteeing bounded response times for temporally constrained traffic which requires temporal consistency among the participating nodes. The most widely accepted time synchronization protocol for Ethernet-based Industrial Communication Protocols is IEEE 1588 [10] [41] which has a distributed implementation that can seamlessly run on Ethernet. The time slotted operation of D^2RIP requires synchronization among nodes. To this end, we employ the widely used IEEE 1588 protocol. This section gives details about timekeeping in OS and explains the synchronization protocol in detail.

4.2.1 Timekeeping Basics

Today's computer operating systems measure the duration of time in two ways: with *tick counting* and *tickless timekeeping* [49]. Tick counting measurement is based on periodic interrupts, called *ticks*, at a known rate. With the help of these ticks, the OS can count how much time has passed. On the other hand, in tickless timekeeping method, OS only reads the clock when needs to learn what time is. Tickless timekeeping has an obvious advantage over timekeeping with ticks due to the low CPU usage to manage timekeeping. In addition, computers or embedded systems should have specific hardware (such as oscillators) to keep track of time. Several clock devices can be found on the same computer:

- PIT (programmable interval timer)
- RTC (real time clock)
- ACPI (advanced programmable interrupt controller)
- TSC (the time stamp counter)
- HPET (the high precision event timer)

TSC and HPET clock sources are much preferred on the current computer systems because of the accuracy and speed to read clock-time. The TSC is a 64-bit cycle counter and can be found on the newer version of Pentium CPUs. The TSC is, by far, the finest grained and most convenient timer device to access. However, it has several drawbacks such as changing of CPU clock speed due to power management. HPET is the newest clock technology and it was developed by Intel. Generally, HPET timer cannot be found in old PC systems and should be activated in bios settings in newer computer systems. The HPET timer has much higher resolution than the others and it is probably the best clock source for D^2RIP.

4.2.2 Clock Source in Linux

From the kernel version *2.6.18*, an abstraction layer called *clocksource* was added to the timekeeping subsystem in Linux. Thus, Linux OS allows the user to choice kernel clock-source. We chose HPET timer with this command:

```
echo 'hpet' >/sys/devices/system/clocksource/clocksource0/current_clocksource
```

Also the maximum frequency of the HPET timer can be set in Linux to increase the accuracy with this command:

```
echo 2048 >/proc/sys/dev/hpet/max-user-freq
```

Most Linux distributions read the initial wall-clock time from the computer’s battery-backed real-time clock when they boot up and query a network time server (like NTP) to obtain a more accurate time value through the process. However, obtaining the time accurately over the long term is challenging because the clock oscillators in computers tend to drift due to temperature changes [50]. Therefore, for long-term clock accuracy, synchronization software should be run in every computer and periodically resynchronize the wall clock time to an external clock. To prevent conflicting changes in the time, one synchronization application should be run at a time in a given computer systems. Therefore, default NTPd synchronization service in Linux should be disabled before using any other synchronization application:

```
sudo /etc/init.d/ntp stop
```

4.2.3 Precision Time Protocol (PTP)

IEEE-1588 standard defines a Precise Time Protocol (PTP) which is the best protocol to synchronize system using standard LAN communication. In PTP, a number of message exchanges take place between a selected master node and the slave nodes periodically as shown in Figure 9. The slave nodes compute their clock offsets from the master node according to these messages and synchronize their clocks with the master node. The maximum offset magnitude can be up to sub-microsecond with hardware time-stamping. Thus, PTP seemed to be the logical decision for synchronizing network based devices, such as D^{2}RIP controllers.

The synchronization principle of the PTP consists of two steps:

1) Step 1 –Offset Measurement
2) Step 2 –Delay Measurement

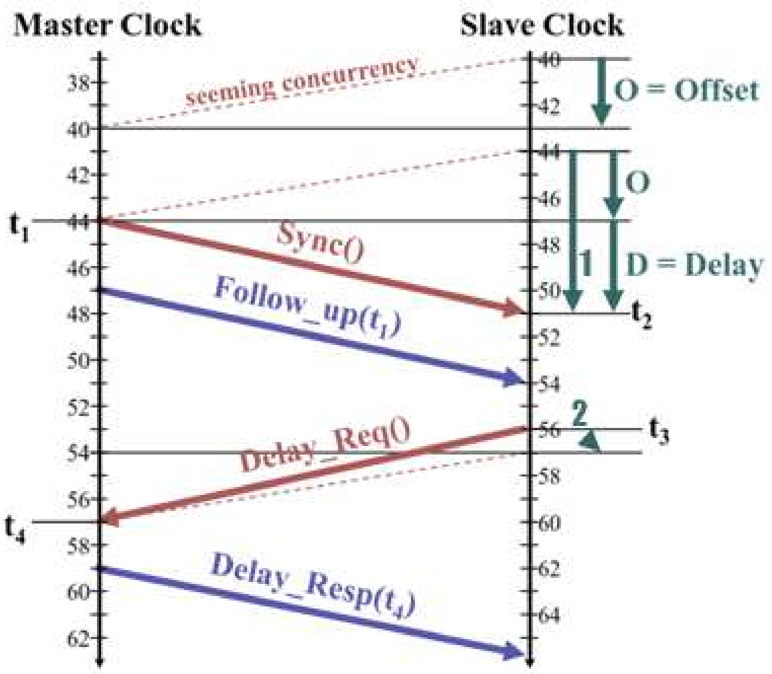


Figure 9 IEEE 1588 Synchronization

$$By\ step\ 1,\ t_2 - t_1 = T_{Delay} + T_{Offset} \tag{4.1}$$

$$By\ step\ 2,\ t_4 - t_3 = T_{Delay} - T_{Offset} \tag{4.2}$$

$$By\ using\ step\ 1\ \&\ 2,\ T_{Delay} = \frac{1}{2}\ [(t_2 - t_1) + (t_4 - t_3)] \tag{4.3}$$

$$By\ using\ step\ 1\ \&\ 2,\ T_{Offset} = \frac{1}{2}\ [(t_2 - t_1) - (t_4 - t_3)] \tag{4.4}$$

$$T_{new\,=}\,T_{old} + T_{Offset} \tag{4.5}$$

Equation 4.5 gives updated clock value for slave. Also, we note that transmit and receive propagation delay in the network are assumed symmetrical.

The accuracy mainly depends on the precision of the time stamps. The possible timestamp locations are shown in Figure 10. Time stamping points should reflect transmit and receive times as precise as possible. In the hardware assisted synchronization, time stamps are taken at the Medium Independent Interface (MII). On the other side, pure software solutions take time stamps in the Network Interface Card (NIC) driver by using *skb_tx_timestamp*() function. Transmission time stamping at the driver level is the best software solution but requires a modified network driver.

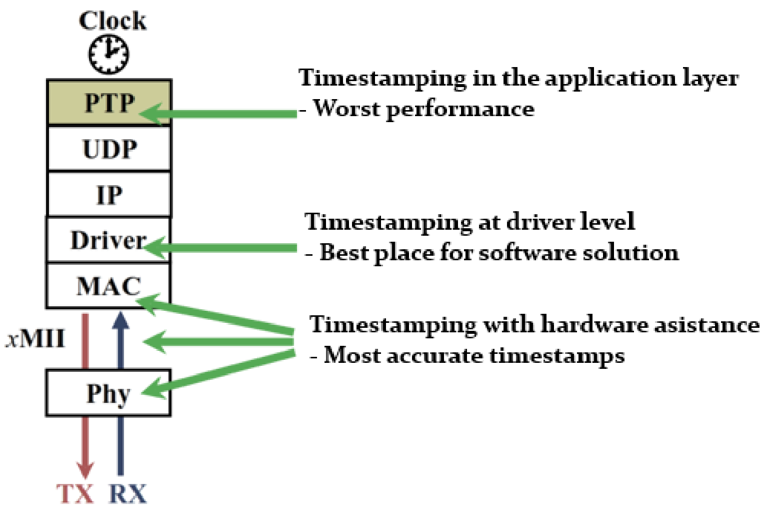

Clock
PTP
UDP
IP
Driver
MAC
xMII
Phy
TX
RX
Timestamping in the application layer
- Worst performance
Timestamping at driver level
- Best place for software solution
Timestamping with hardware asistance
- Most accurate timestamps

Figure 10 Possible Timestamp Locations

Clock synchronization without specific hardware equipment leads to larger errors and consequently, low accuracy. Moreover, system load and intensive disk I/O result in the huge delay in system time with software synchronization. Therefore, software clock synchronization is not suitable method to obtain high time precision when dealing with RT communications. Hardware time stamping is the only way to reach high accuracy. Only some Ethernet adaptor manufactures support hardware time stamping on their devices. For example, such devices are the Intel 82576, 82574, 82583, 82599 gigabit Ethernet controllers.

The accuracy of the PTP protocol also depends on the latency jitter of the network topology. Hubs and point to point connections provide the best solution. However, switches cause high delay and jitter due to the storing of the incoming packets. The problem can be solved with the usage of IEEE 1588 Boundary clocks in switches. We only use 100 Mbit/s hub connections for D^{2}RIP and there is nearly no delay jitter expected between master clock and slave clocks because of no internal queuing delay.

The significant problem is how Linux kernel clock should be synchronized with external PTP hardware clock. PTP hardware clock cannot be set as a main clock source in Linux OS due to its negative aspects. The idea of using the PTP hardware clock to the Linux kernel as a main clock source was considered but ultimately rejected, as discussed in [51]. Therefore, Linux system clock should be synchronized with PTP clock with acceptable accuracy. The best idea, *the two-level PTP method*, is somewhat similar to using a PPS (Pulse Per Second) to synchronize the Linux system clock to the PTP clock, illustrated in Figure 11 [52]. For this method, at least two applications should be run on the system. The first application synchronizes the PTP slave hardware clock to the PTP master hardware clock over the Ethernet by sending synchronization packets, and the second application adjusts the system time (Wall clock time) to the PTP slave hardware clock by using proportional-integral (PI) controller [52].

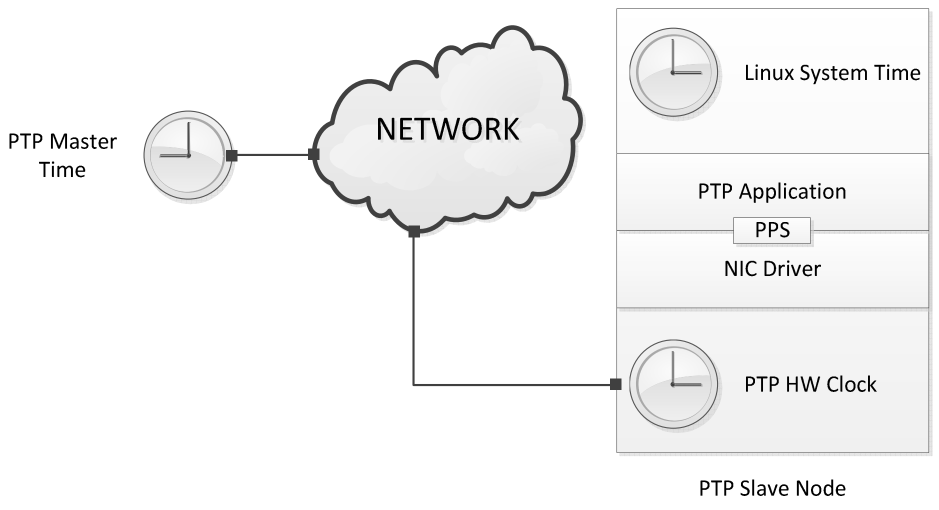


Figure 11 Synchronizing the System Clock to the PTP Clock

Another problem is how the PTP clock is used in Linux and hardware time stamping can be reached. The Linux network stack already supports hardware time stamping since Linux version 2.6.30. A PTP clock driver registers itself with the class driver in Linux OS. This class driver handles all of the connection with the user space. The PTP class driver creates a character device named generally */dev/ptp0*. User space applications can access directly PTP clock driver by using standardized input/output control (ioctl) methods such as open, read etc. Obtaining timestamps on an open socket requires a two main step. Firstly, the Ethernet driver must be configured for hardware time stamping using the ioctl method *SIOCSHWTSTAMP* [51]. Secondly, the synchronization applications request the socket option named *SO_TIMESTAMPING*. *SO_TIMESTAMPING* socket option is for packet time stamping for PTP. It allows enabling receive or transmit packet timestamps, and also allows a software fallback if hardware time stamping should be unavailable [51]. Once the socket is properly initialized, the *recvmsg()* call will return a control message containing the timestamp along with the packet data. For outgoing packets, the packet will be looped back to the socket’s error queue, where the packet reappears with the timestamp control message added [51].

Our IEEE 1588 protocol implementation is carried out by an existing software implementation of IEEE 1588 [53] in addition to the driver and NIC support for IEEE 1588. This software is fully compliant with the IEEE 1588-2008 standard and supports hardware time stamping and PTP Hardware Clock (PHC) to achieve high precision. It uses the *SO_TIMESTAMPING* socket option that is standard in Linux. This software carries out the following:

- Sending IEEE 1588 packets to exchange the delay and offset information
- Among nodes for synchronization
- Adjusting the PHC
- Synchronizing the system clock (REALTIME_CLOCK) and PHC

4.2.4 Requirements of the Hardware Time Stamping

There are 3 requirements to use hardware time stamping in strict sense:

1) A suitable NIC adaptor that supports hardware time stamping:
   We use the Intel Gigabit CT Desktop Adapter [54] in every D²RIP controller because of its availability at low cost. It uses Intel 82574L Ethernet controller which fully supports hardware time stamping. The card is installed in the PCI express slot on the mainboard and there is no further hardware requirement. It uses the e1000e driver module with version 2.3.2 that was published in March 2013 [55].

   Starting with version 3.5 of the Linux kernel, the time stamping capabilities of a network card can be queried using the *ETHTOOL_GET_TS_INFO* ioctl. Using standard Ethernet tool named *ethtool* [56], we can query the capabilities of our NIC card:

   ```
   ethtool -T eth0
   ```

2) Linux kernel that supports PTP Hardware Clock (PHC) subsystem:
   Two features are required to support hardware time stamping (PTP): *network packet time stamping* and *clock control*. Linux Kernel versions 2.6.30 and above possess the *SO_TIMESTAMPING* option that enables a driver module in the kernel space to send the timestamp values to the IEEE 1588 software that runs in the user space. Hence, this option fulfills the network packet time stamping requirement. The clock control requirement is fulfilled with the PHC support that is available on kernel versions 3.0 and above. It provides an API for user space management of the hardware clock in the kernel space. Furthermore, certain settings are required in the Linux kernel which is listed in Section 4.1. Therefore, we use Linux kernel version *3.6.2*.

   Some settings should be reconfigured in Linux kernel to activate hardware time stamping:

   - Enable “Timestamping in PHY devices”
   - Enable “PTP Hardware Clock (PHC)”
   - Enable “PTP clock support”

3) Ethernet driver that supports hardware time stamping and PTP clock driver:
   Ethernet driver should support hardware time stamping and PTP hardware clock driver. Intel Gigabit CT Desktop Adapter uses *e1000e* module as driver. The hardware timestamp support for *e1000e* module came with driver version *2.2.14*. We use *e1000e* driver version *2.4.14* in Linux OS.

By default, hardware time stamping is not enabled by *e1000e* driver. User should recompile the source code of the driver to activate the property with this command:

```
make CFLAGS_EXTRA=-DE1000E_PTP
```

The default *e1000e* driver should be removed from the Linux operating system and loaded the re-compiled driver with these commands:

```
rmmod e1000e
insmod e1000e.ko
```

4.3 D²RIP IO plug-in for libFAUDES

The operation of the workcell and the distributed controllers is simulated using the simulator plug-in that is part of the libFAUDES discrete event systems software library developed at the University of Erlangen-Nuremberg [57]. The D²RIP IO plug-in provides a libFAUDES interface to the D²RIP, to be used e.g. for the synchronization of events among multiple instances of simfaudes for the decentralized supervision of discrete event systems. This plug-in provides data-types to support the automatic execution of discrete event systems. It serves as a basis for applications that perform a hardware-in-the-loop simulation of a controller interacting with a physical plant, or perform an stochastic performance analysis. In order to obtain realistic behavior in our application example as explained in section 3.7, we use stochastic execution attributes for the events *stpR*, *stpC* and *fPD* as specified by the below configuration file:

```
<SimEventAttributes>
        "stpR"
        <Stochastic>
                +Trigger+     +Gauss+
                <Parameter>
                        10   20  15  20
                </Parameter>
        </Stochastic>
</SimEventAttributes>
```

The D²RIP IO plug-in can be configured with a XML file according the communication requests related to the controller's input and output events:

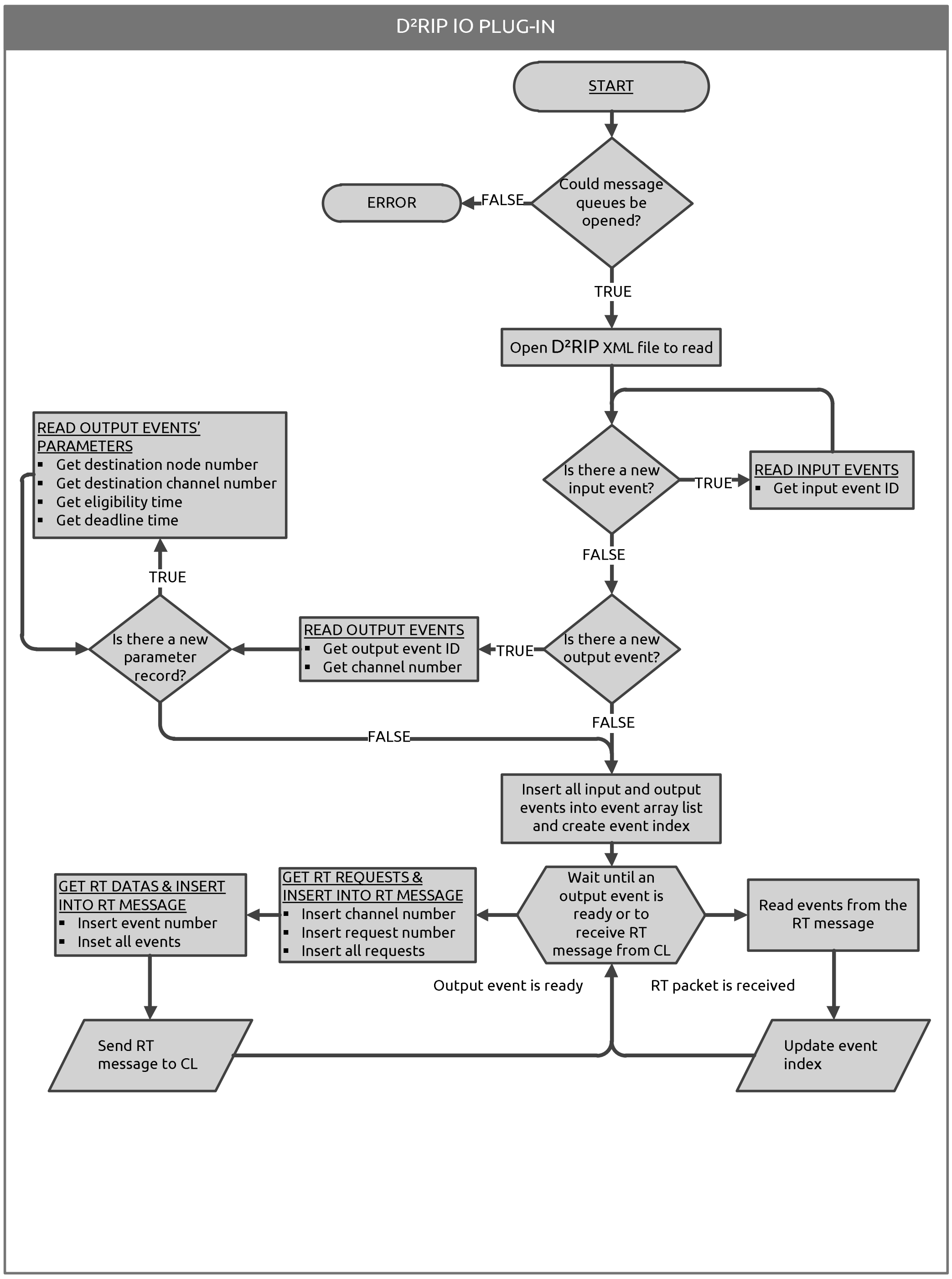
```
<Event name="mvC?" iotype="output">
        <EventId value="1"/>
        <ChannelToTransmit value="1"/>
        <ParameterRecord>
                <DestinationNode value="2"/>
                <DestinationChannel value="1"/>
                <EligibilityTime value="4" />
                <DeadlineTime value="5"/>
        </ParameterRecord>
</Event>
```

```
<Event name="mvC!" iotype="input">
        <EventId value="2"/>
</Event>
```

The libFAUDES source code should be downloaded and compiled with the D²RIP IO plug-in source code to build D²RIP IO plug-in. In libFAUDES's Makefile, "timed", "simulator" and "iodevice" plug-ins should be enabled and in "iodevice" Makefie, "iop_simplenet" and "iop_d3ripURT" plug-ins should be enabled by removing to comment symbol to build D²RIP IO plug-in properly. Now, libFAUDES sources can be re-configured and compiled with the commands:

```
make dist-clean
make configure
make
```

The flowchart of the D²RIP IO plug-in is shown in Figure 12.



D²RIP IO PLUG-IN
START
Could message queues be opened?
FALSE
ERROR
TRUE
Open D²RIP XML file to read
Is there a new input event?
TRUE
READ INPUT EVENTS
▪ Get input event ID
FALSE
Is there a new output event?
TRUE
READ OUTPUT EVENTS
▪ Get output event ID
▪ Get channel number
Is there a new parameter record?
TRUE
READ OUTPUT EVENTS' PARAMETERS
▪ Get destination node number
▪ Get destination channel number
▪ Get eligibility time
▪ Get deadline time
FALSE
FALSE
Insert all input and output events into event array list and create event index
Wait until an output event is ready or to receive RT message from CL
GET RT REQUESTS & INSERT INTO RT MESSAGE
▪ Insert channel number
▪ Insert request number
▪ Insert all requests
GET RT DATAS & INSERT INTO RT MESSAGE
▪ Insert event number
▪ Inset all events
Send RT message to CL
Output event is ready
RT packet is received
Read events from the RT message
Update event index

Figure 12 D²RIP IO Plug-in Flowchart

4.4 Time Slots, Encapsulation and Data Structure

The duration of the time slot *dSlot* is an important parameter that affects the RT response of D²RIP as demonstrated in the example in Section 3.7. The control application progresses with the time slots as shown in Figure 6. Hence, the rate of completing tasks in the application increases as *dSlot* decreases. This result is quantified in (3.2) Section 3.6 which states that to meet the message deadlines the time slot duration should be smaller than a bound. In other words, *dSlot* determines the deadlines that can be guaranteed by D²RIP for a given application. To this end, we selected the OS and its configurations as listed in Section 4.1 to shorten *dSlot* as much as possible provided that it is long enough to carry the longest RT message. Furthermore, as stated in Section 2.1.2 a high precision IEEE 1588 time synchronization with hardware support is adopted to minimize the time slot guard times. Every controller in D²RIP should wait until the beginning of a new slot when they have finished the previous slot earlier. It is very crucial to sleep precisely and wake up at the same time when a new slot begins. Therefore, high precise sleep with accuracy in microseconds is very important for the slot duration. Standard Unix functions like *nanosleep()* and *clock_nanosleep()* are incapable for high precision. Therefore, the time required to sleep should be divided into parts as shown in Figure 13.

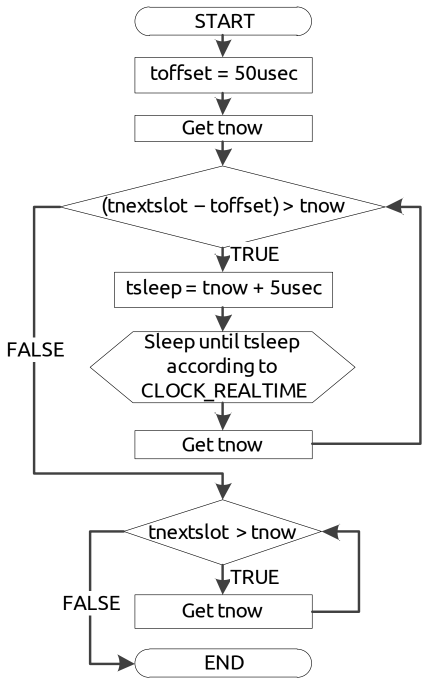


Figure 13 High Precision Sleep with Partially Sleep and Busywait

The fixed time slot duration *dSlot* as introduced in Chapter 3 requires a bound on the maximum length frames that are sent by IL on the shared medium. We denote this bound $Frame_{max}$ and select its current value as 504 Bytes including 14 Bytes of Ethernet Header and 4 Bytes of CRC. We use the `EtherType` numbering field in the Ethernet frame to indicate the type of data carried in the Ethernet frame. The value of this field is *0x1100* for RT and *0x2200* for the fragments of an nRT packet that exceeded the $Frame_{max}$. Any other `EtherType` value indicates a short nRT packet that was not fragmented. The payload of this fixed Ethernet frame can be either a CL_PACKET, or an nRT_PACKET which is a short nRT application packet from standard upper layer protocols or an nRT_FRAGMENT that is constructed from a long nRT packet. These three encapsulation possibilities are depicted in Figure 14, Figure 15 and Figure 16.

The IL_PACKET as depicted in Figure 14 has an IL_PACKET_HEADER that consists of a 16 Byte time value that is used for timestamp information for RT packet. Also CL_PACKET payload consists of an 8 Byte timestamp information. These time values are valid only for testing purpose. Therefore, these bytes are not taken into account when calculations are performed.

The CL_PACKET as depicted in Figure 14 has a CL_PACKET_HEADER that consists of an 8 Byte time value that is used for the protocol initialization, a 4 Byte slot number and a 2 Byte Packet length field that carries the length of the entire CL_PACKET.

CL_PACKET payload is RT_APP_MESSAGE and can be empty if there is no such message to send from node *i* in the current time slot. The length of the RT_APP_MESSAGE is limited to $Frame_{max} - ((14+4) + (8+4+2))$ Bytes. The data that is generated by the control application consists of communication requests as described in Section 3.3 that are provided for the dynamic bandwidth allocation and further control application related data. The RT_APP_MESSAGE begins with a 1 Byte field NoR that shows the number of communication requests that are sent in this message. As described in Section 3.3 each request consists of a (b, c, eT, dT) tuple, whereby 1 Byte is reserved for each tuple entry. In this context, we assume that *dSlot* (and consequently $\mathrm{Frame}_{\mathrm{max}}$) is selected such that no fragmentation is required for the longest possible CL payload. This assumption is not a restriction of generality, since the length of each RT message is known because of the deterministic application model. For example, the workcell as described in Section 3.7 only requires sending one communication request and 3 Bytes of control application data in RT messages. Hence, a payload of 8 Bytes is sufficient.

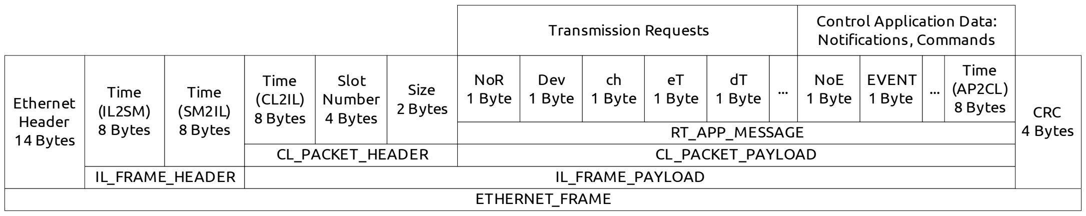


Figure 14 RT Encapsulation in D²RIP

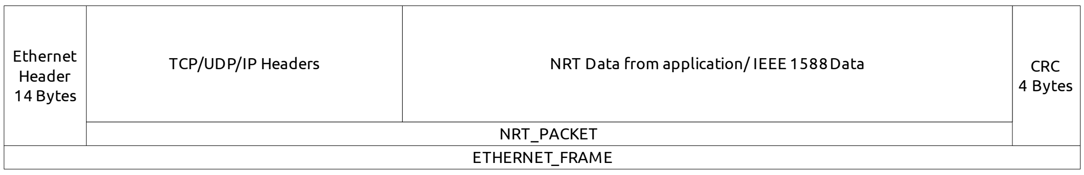


Figure 15 nRT Encapsulation in D²RIP

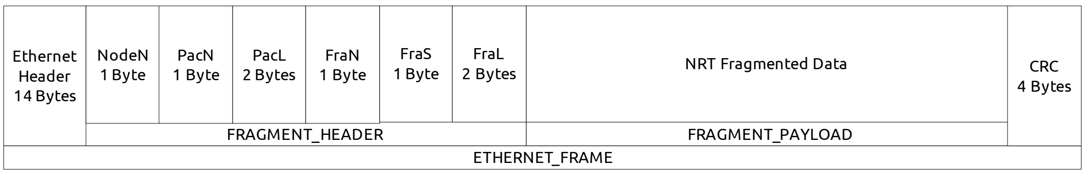


Figure 16 nRT Fragment Encapsulation in D²RIP

4.5 Fragmentation and Reassembly of nRT Frames

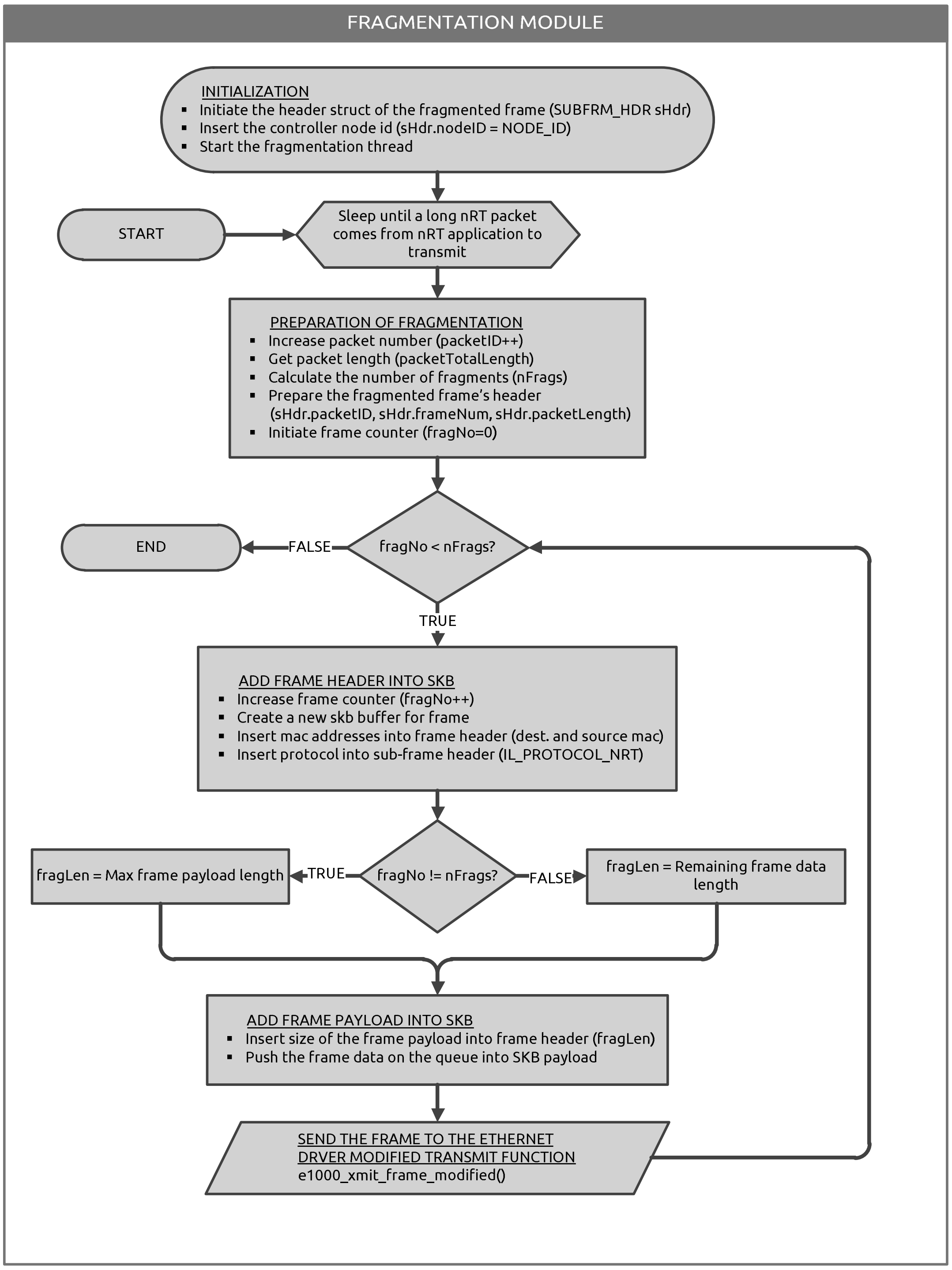
The nRT packets that can be carried in an Ethernet frame with a size less than $Frame_{max}$ are carried as ordinary Ethernet payloads. The corresponding encapsulation is shown in Figure 15.

The nRT packets that are too long to be transmitted in a single time slot are processed by two kernel threads, *Fragmentation* and *Reassembly*. A header structure that is appended to the nRT packets carries relevant information for the operation of these threads similar to the header fields in IP packets that are relevant for fragmentation. The header structure is composed of: sending node (NodeN), a unique packet identification (PacN), packet length before fragmentation (PacL), total number of fragments (FraN), the fragment sequence (FraS) in the packet before fragmentation and the fragment length (FraL) as seen in Figure 16. The nRT packets in D²RIP are fragmented only once by the sender node if necessary.

In addition, we implemented a second transmit function for the e1000e driver module that we call *e1000_xmit_frame_modified()* which intercepts the packets coming from the upper layer and wakes up the *Fragmentation thread* when a long nRT_PACKET arrives. This thread first prepares and initializes the header data structure as defined above, then fragments the data in nRT_PACKET, sets the fields of the fragmentation header and then encapsulates the resulting packet in an Ethernet Frame with a maximum size of $Frame_{max}$.

The *Reassembly thread* keeps an array of buffers *reassembly_pkt_array[sHdr.nodeID].skb* and it wakes up when a fragmented nRT packet is received. It copies the received fragments while FraN>1 and FraS<FraN in the respective buffer in this array corresponding to the sending node. When FraO = FraN, then the received packet is forwarded to the user space with *netif_receive_skb()*. *netif_receive_skb()* is the main Ethernet receive function in

Linux. If the reassembled packet length does not match PacL, the Reassembly discards the packet. The flowcharts of the fragmentation and reassembly module are shown in Figure 17 and Figure 18.



FRAGMENTATION MODULE
INITIALIZATION
Initiate the header struct of the fragmented frame (SUBFRM_HDR sHdr)
Insert the controller node id (sHdr.nodeID = NODE_ID)
Start the fragmentation thread
START
Sleep until a long nRT packet comes from nRT application to transmit
PREPARATION OF FRAGMENTATION
Increase packet number (packetID++)
Get packet length (packetTotalLength)
Calculate the number of fragments (nFrags)
Prepare the fragmented frame's header (sHdr.packetID, sHdr.frameNum, sHdr.packetLength)
Initiate frame counter (fragNo=0)
END
FALSE
fragNo < nFrags?
TRUE
ADD FRAME HEADER INTO SKB
Increase frame counter (fragNo++)
Create a new skb buffer for frame
Insert mac addresses into frame header (dest. and source mac)
Insert protocol into sub-frame header (IL_PROTOCOL_NRT)
fragLen = Max frame payload length
TRUE
fragNo != nFrags?
FALSE
fragLen = Remaining frame data length
ADD FRAME PAYLOAD INTO SKB
Insert size of the frame payload into frame header (fragLen)
Push the frame data on the queue into SKB payload
SEND THE FRAME TO THE ETHERNET DRVER MODIFIED TRANSMIT FUNCTION
e1000_xmit_frame_modified()

Figure 17 Fragmentation Flowchart

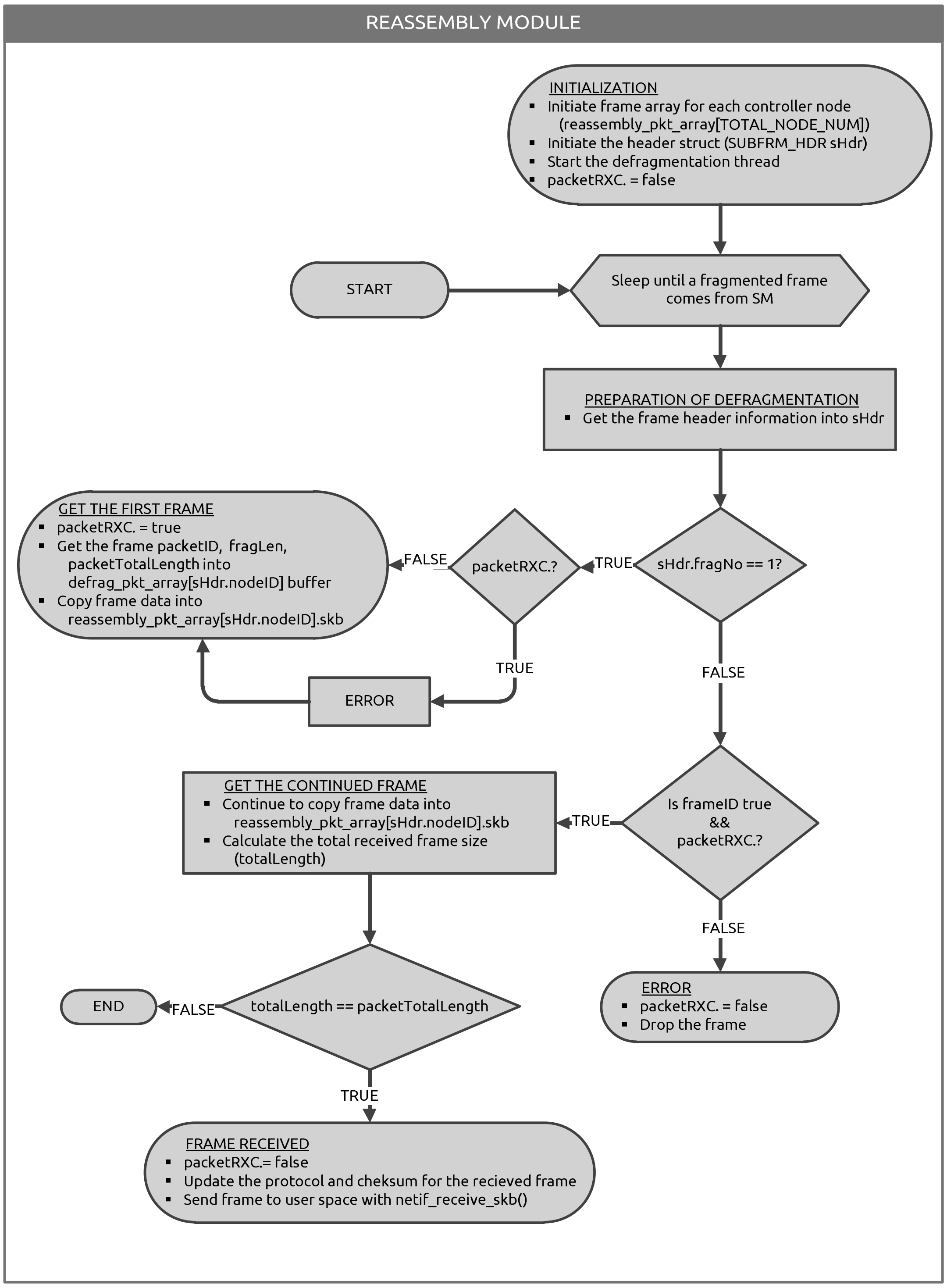

REASSEMBLY MODULE
INITIALIZATION
Initiate frame array for each controller node (reassembly_pkt_array[TOTAL_NODE_NUM])
Initiate the header struct (SUBFRM_HDR sHdr)
Start the defragmentation thread
packetRXC. = false
START
Sleep until a fragmented frame comes from SM
PREPARATION OF DEFRAGMENTATION
Get the frame header information into sHdr
sHdr.fragNo == 1?
TRUE
packetRXC.?
FALSE
GET THE FIRST FRAME
packetRXC. = true
Get the frame packetID, fragLen, packetTotalLength into defrag_pkt_array[sHdr.nodeID] buffer
Copy frame data into reassembly_pkt_array[sHdr.nodeID].skb
TRUE
ERROR
FALSE
Is frameID true && packetRXC.?
TRUE
GET THE CONTINUED FRAME
Continue to copy frame data into reassembly_pkt_array[sHdr.nodeID].skb
Calculate the total received frame size (totalLength)
FALSE
ERROR
packetRXC. = false
Drop the frame
totalLength == packetTotalLength
FALSE
END
TRUE
FRAME RECEIVED
packetRXC.= false
Update the protocol and cheksum for the recieved frame
Send frame to user space with netif_receive_skb()

Figure 18 Reassembly Flowchart

4.6 Initialization of D^{2}RIP

At system start-up, D^{2}RIP needs to establish suitable synchronization accuracy before simultaneously entering the time-slotted operation. The IL of D^{2}RIP distinguishes the cases of initialization and a running stack. Our IL implementation keeps a boolean variable D^{2}RIP_Init where a true value for this variable indicates that the stack is running. Here we would like to note that one of the controller acts as a master node temporarily to send the first message. When IL receives a CL_PACKET with Slot Number = 1, D^{2}RIP_Init is set to true and the modification on the Ethernet Driver is activated. If IL receives a 1 Byte CL Payload data where CLP = 0xFF then D^{2}RIP_Init is reset to *false* and the modification on the Ethernet Driver is deactivated. The CL thread sleeps until the IEEE 1588 synchronization is established and wakes up after a certain time following the reception of the first synchronization message.

4.7 Coordination Layer Implementation

The coordination layer is implemented as a thread in the user space according to the data structure for CL that is presented in Section 3.3. For all nodes *i*, the thread that runs in CL_i wakes up periodically at the beginning of each time slot. In each time slot, the processes related to the transmission and reception of an RT message by all of the nodes on the shared medium take place. In our D^{2}RIP implementation the timing reference is the CLOCK_REALTIME because of system clock and hence it triggers the wake up of the CL thread.

When the CL thread wakes up in node *i*, it first executes AP2CL as described in Section 3.3. If the application has a message to send, then the received RT_APP_MESSAGE is inserted in Tx_i. Next, CL_i updates the decision variables $RTCL_i$, $myCL_i$ and $myCH_i$ according to the first eligible request in the priority queue of requests PQ_i as described in Section 3.3. Then,

- if $myCL_i$ = false then the RT_APP_MESSAGE is empty and CL_i notifies IL_i by sending 1 Byte of information with these decision variables.

- if $RTCL_i$ = true and $myCL_i$ = true then node *i* is expected to send the first RT message that is stored in Tx_i buffer for $myCH_i$. It is possible that the control application is waiting for some event to happen and hence no RT message is produced and Tx_i is empty. In this case the $RTCL_i$ = true and $myCL_i$ = true variables are sent to IL without a message via CL2ILRT. If Tx_i is non-empty, then the RT_APP_MESSAGE is dequeued and sent to IL_i via the character device. The transmission of the message then takes place on the shared medium as described in Section 4.8.

When IL_i receives a frame from the shared medium in an RT slot, it forwards the encapsulated CL_PACKET to CL_i as described in Section 4.8. If there is an RT_APP_MESSAGE in the received packet, then the transmission requests are extracted and inserted in the priority queue of requests PQ_i.

Here we would like to note that as described in Section 3.5, the requests are inserted in the priority queue when a CL_PACKET is received. Hence, it is required that the sender node also receives the message to have the consistent state of the CL. While the broadcast on a true shared medium ensures that, if the nodes are connected by a physical layer hub, the message is repeated on all ports except for the port that it is received from. To this end, we add the configurable ECHO_SENDER option which loops the sent CL_PACKETs back to CL_i if the implementation medium contains any network devices that disrupt the broadcast including the sender.

The RT control applications send their RT messages to the CL by calling AP2CL $(dat, par, ch)_i$ along with the channel information and the control message. It contains 1 Byte channel information, so that CL puts the received control message into the corresponding transmit buffer (i.e.: $Tx_i[ch]$). Note that the CL data structure presented in Section 3.3 puts the message to the transmit buffer without extracting the communication requests.

Both the control application and CL implementation are in the user space and they exchange messages via 2 message queues MQUEUE_AP2CL and MQUEUE_CL2AP. The reads by CL conducted on MQUEUE_AP2CL are non-blocking where CL goes on its operation if there is no application data ready in the queue to keep *dSlot* as short as possible. The control application requires the data that arrives from CL as soon as possible hence, the application reads MQUEUE_CL2AP queue in the blocking mode, waiting for the CL data to arrive to go on its operation.

In Section 3.3, we defined, PQ_i.Top as the pointer to the request (b, c, ET, DT) with the smallest absolute deadline that is at the same time eligible. This request determines if the current time slot is RT or nRT and the ownership of the RT slots. The priority queue is updated with the arrival of each new CL_PACKET.

We implement the PQ_i such that it stores all of the eligible requests that are ordered according to their deadline, hence PQ_i.Top holds the eligible request with the smallest deadline. A second priority queue PQ_i.E stores the requests that are not eligible. Hence for a given absolute time t, $ET > t$ holds for all requests in PQ_i.E. PQ_i.E.Top is the request with the smallest *ET*. A requests that is received at time t in a CL_PACKET, is not eligible at time *t* as $eT > 0$ and hence $ET > t$. Consequently, each new request first inserted in PQ_i.E. At the beginning of each slot time *t*, CL first checks if the request at PQ_i.E.Top has $ET \leq t$. If so, that request is dequeued and inserted in PQ_i according to its *DT* and PQ_i.E. Top is updated. Then CL decides about the time slot by peeking at the request at PQ_i.Top position. PQ_i is updated once every slot when new requests are received in a CL_PACKET by inserting them in PQ_i.E and dequeuing PQ_i.Top. Both priority queues PQ_i and PQ_i.E are implemented as binary heaps.

The flowchart of the CL thread is shown in Figure 19.

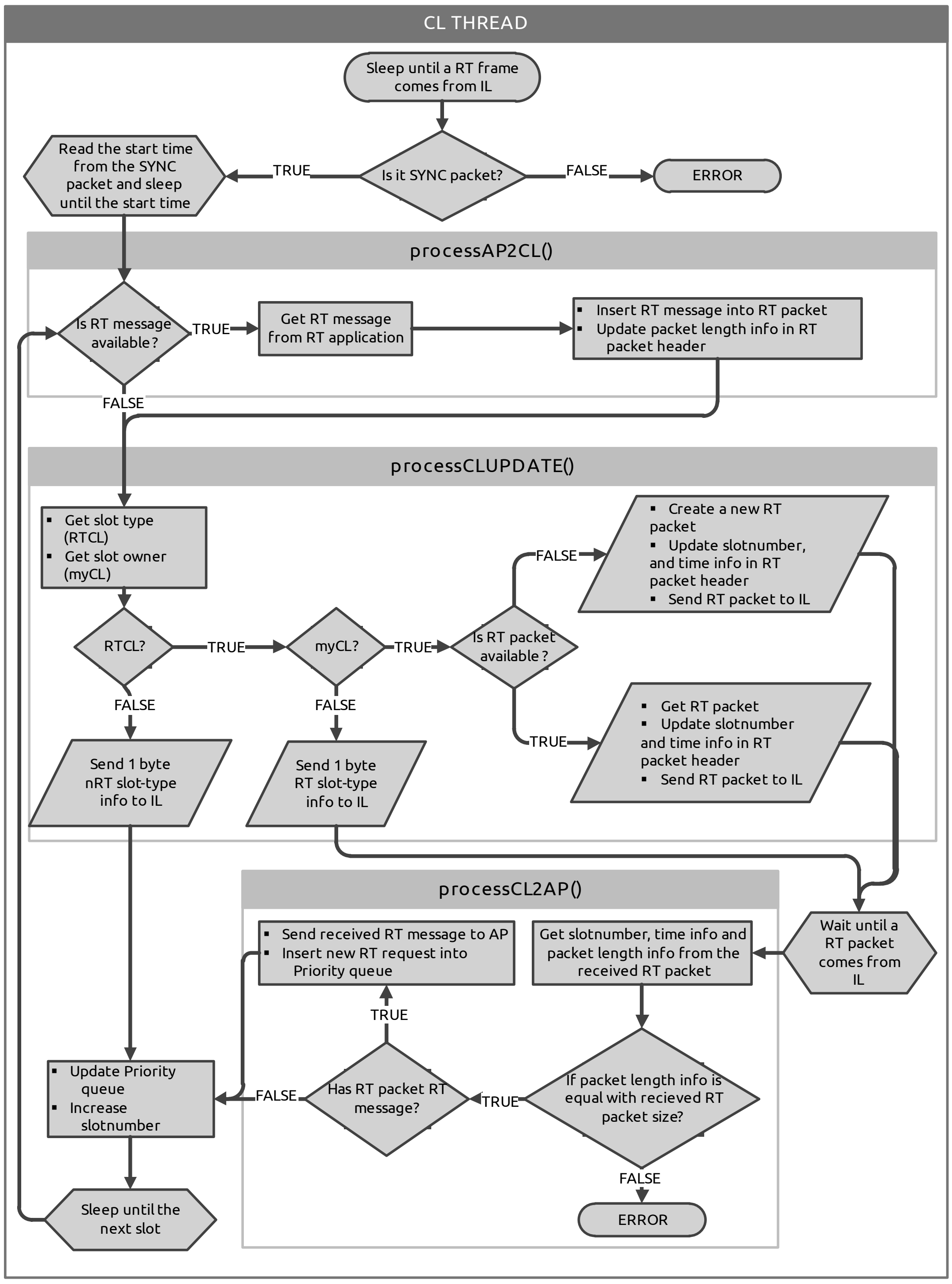

CL THREAD
Sleep until a RT frame comes from IL
Is it SYNC packet?
TRUE
FALSE
ERROR
Read the start time from the SYNC packet and sleep until the start time
processAP2CL()
Is RT message available?
TRUE
Get RT message from RT application
Insert RT message into RT packet
Update packet length info in RT packet header
FALSE
processCLUPDATE()
Get slot type (RTCL)
Get slot owner (myCL)
RTCL?
TRUE
myCL?
TRUE
Is RT packet available?
FALSE
Create a new RT packet
Update slotnumber, and time info in RT packet header
Send RT packet to IL
TRUE
Get RT packet
Update slotnumber and time info in RT packet header
Send RT packet to IL
FALSE
Send 1 byte nRT slot-type info to IL
FALSE
Send 1 byte RT slot-type info to IL
processCL2AP()
Wait until a RT packet comes from IL
Get slotnumber, time info and packet length info from the received RT packet
If packet length info is equal with recieved RT packet size?
TRUE
FALSE
ERROR
Has RT packet RT message?
TRUE
Send received RT message to AP
Insert new RT request into Priority queue
FALSE
Update Priority queue
Increase slotnumber
Sleep until the next slot

Figure 19 CL Thread Flowchart

4.8 Interface Layer Implementation

The Interface Layer is responsible for selectively transmitting the RT and nRT packets in the current time slot according to the information received from CL. We indicate the IL instance that runs on node *i* as IL_i as in Section 3.4. Note that there can be multiple Ethernet interfaces on a node as we describe in our experiment in Chapter 5. Hence, we denote the interface that runs D^2RIP as eth0. Currently, we use the e1000e Ethernet driver module for eth0.

The IL implementation consists of 3 kernel space modules; Fragmentation, Reassembly and IL module. The IL module implements the actions CL2ILRT and IL2SM as described in Section 3.4. Furthermore the controlled transmission of RT and nRT packets requires modification on the Ethernet driver e1000e. Fragmentation and Reassembly are kernel modules implemented as threads which sleep when they are not executed. When these modules are loaded together with the modified e1000e driver, only Ethernet frames with a maximum size of $Frame_{max}$ are transmitted on the shared medium.

The action CL2ILRT passes the decision parameters $RTCL_i$, $myCL_i$ from CL_i to IL_i together with a possible RT_APP_MESSAGE as described in Section 3.4. If RT_APP_MESSAGE is not empty, then node *i* has RT data to transmit and for the current slot $RTIL_i$ = true and $myIL_i$ = true. If RT_APP_MESSAGE is empty then only 1 Byte of $RTCL_i$, $myCL_i$ is passed to IL. If $RTIL_i$ = true then consequently $myIL_i$ = false. If $RTIL_i$ = false then $myIL_i$ = nRTslotOwner() where *nRTslotOwner()* returns the ownership of the nRT slot according to the cyclically repeating slot assignment as described in Section 3.4.

The $TxnRT_i$ buffer in the IL data structure (as described in Section 3.4) is implemented in the form of two FIFO buffers: $TxnRTH_i$ and $TxnRT_i$. The IEEE 1588 packets are transmitted as nRT packets as described in Chapter 3. To this end, $TxnRTH_i$ stores the high priority IEEE 1588 synchronization messages and $TxnRT_i$ stores the remaining nRT messages. All fragmented and short nRT packets are inserted into the respective FIFO queue upon the function call AP2ILNRT. When node *i* has the right to transmit an nRT packet, it is first checked if there is a synchronization frame in $TxnRTH_i$. If $TxnRTH_i$ is empty, node *i* transmits a possibly fragmented nRT packet if $TxnRT_i$ is non-empty.

IL_i transmits the RT or nRT frames according to the information that is received from CL_i. The CL_PACKETs that are received from CL_i and the nRT PACKETs that are in $TxnRT_i$ and $TxnRTH_i$ are encapsulated and sent to the standard transmit function *e1000_xmit_frame()* implementing the action IL2SM described in Section 3.4.

We implement SM2IL as described in Section 3.4 for different cases. Consider that a frame is received on the eth0 interface. First, if `EtherType` of the frame indicates that it is a nRT_PACKET fragment, then it is forwarded to the Reassembly thread. After the reassembly is complete, the frame is passed on to the user space with *netif_receive_skb()*

which hands off the socket buffer to the upper layers. Second, if the frame is a short nRT_PACKET then it is directly passed to the standard receive function *e1000_receive_frame()*. Third, if the EtherType of the frame indicates that it is a CL PACKET, then it is passed to IL2CLRT which then forwards the packet to the CL implementation in the user space.

The flowchart of the IL thread is shown in Figure 20 and the flowcharts of the transmission algorithm and reception algorithm are shown in Figure 21 and Figure 22.

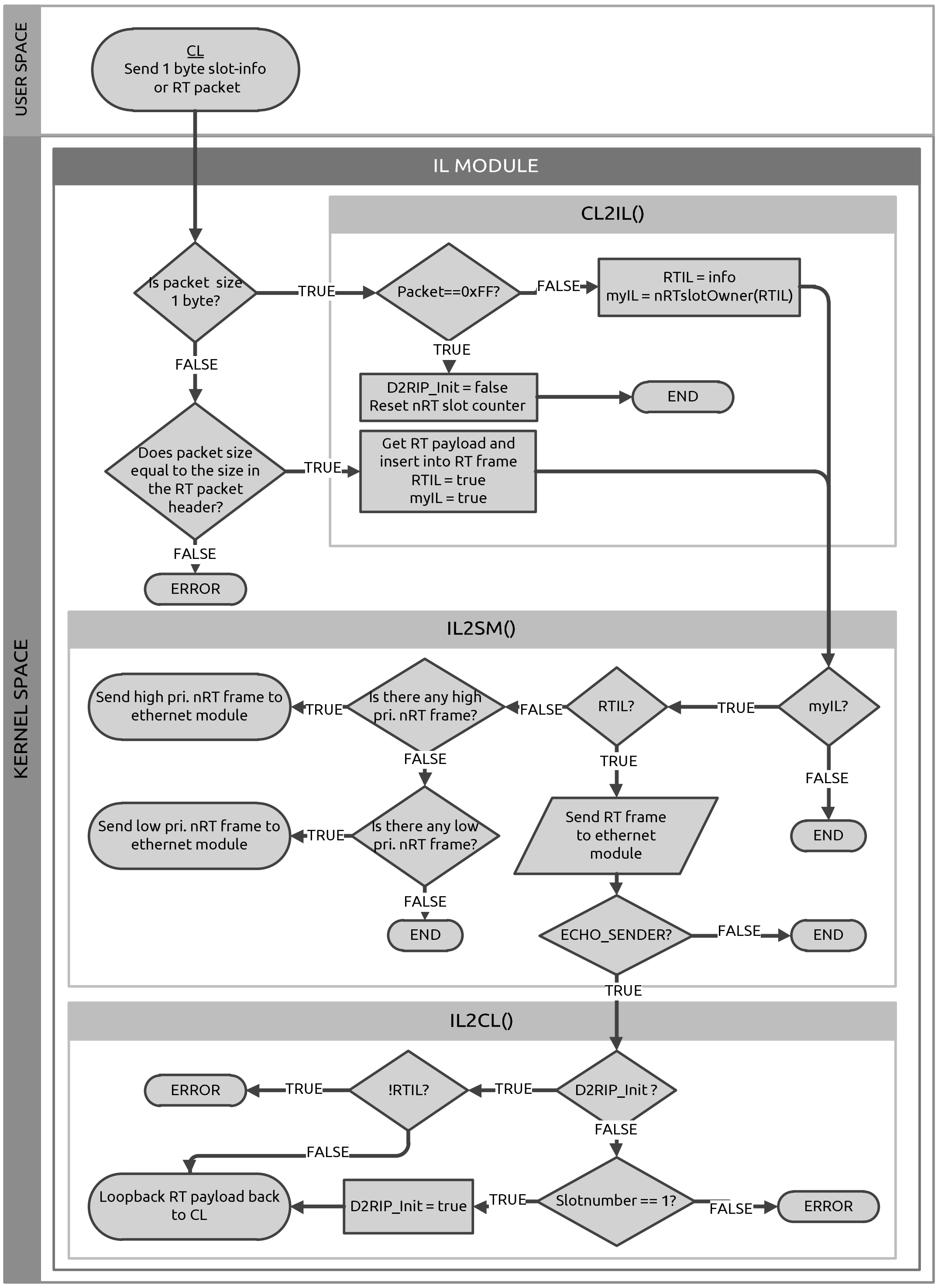

USER SPACE
CL
Send 1 byte slot-info or RT packet
KERNEL SPACE
IL MODULE
CL2IL()
Is packet size 1 byte?
TRUE
Packet==0xFF?
FALSE
RTIL = info
myIL = nRTslotOwner(RTIL)
TRUE
D2RIP_Init = false
Reset nRT slot counter
END
FALSE
Does packet size equal to the size in the RT packet header?
TRUE
Get RT payload and insert into RT frame
RTIL = true
myIL = true
FALSE
ERROR
IL2SM()
Send high pri. nRT frame to ethernet module
TRUE
Is there any high pri. nRT frame?
FALSE
RTIL?
TRUE
myIL?
FALSE
TRUE
FALSE
END
Send low pri. nRT frame to ethernet module
TRUE
Is there any low pri. nRT frame?
Send RT frame to ethernet module
FALSE
END
ECHO_SENDER?
FALSE
END
TRUE
IL2CL()
ERROR
TRUE
!RTIL?
TRUE
D2RIP_Init ?
FALSE
FALSE
Loopback RT payload back to CL
D2RIP_Init = true
TRUE
Slotnumber == 1?
FALSE
ERROR

Figure 20 IL Module Flowchart

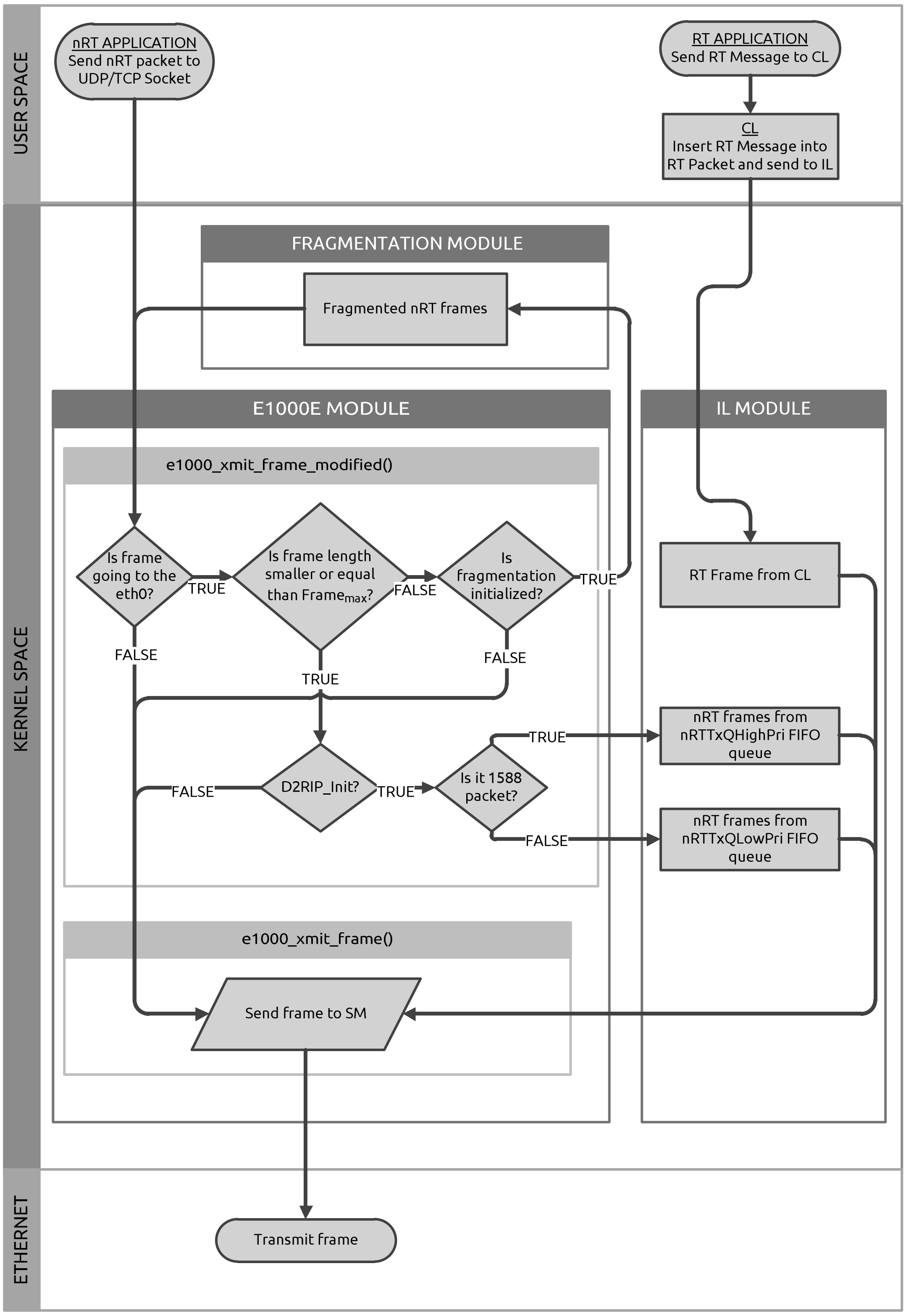

USER SPACE
KERNEL SPACE
ETHERNET
nRT APPLICATION
Send nRT packet to UDP/TCP Socket
RT APPLICATION
Send RT Message to CL
CL
Insert RT Message into RT Packet and send to IL
FRAGMENTATION MODULE
Fragmented nRT frames
E1000E MODULE
IL MODULE
e1000_xmit_frame_modified()
Is frame going to the eth0?
TRUE
FALSE
Is frame length smaller or equal than Frame$_{max}$?
FALSE
TRUE
Is fragmentation initialized?
TRUE
FALSE
RT Frame from CL
D2RIP_Init?
FALSE
TRUE
Is it 1588 packet?
TRUE
FALSE
nRT frames from nRTTxQHighPri FIFO queue
nRT frames from nRTTxQLowPri FIFO queue
e1000_xmit_frame()
Send frame to SM
Transmit frame

Figure 21 Transmission Flowchart

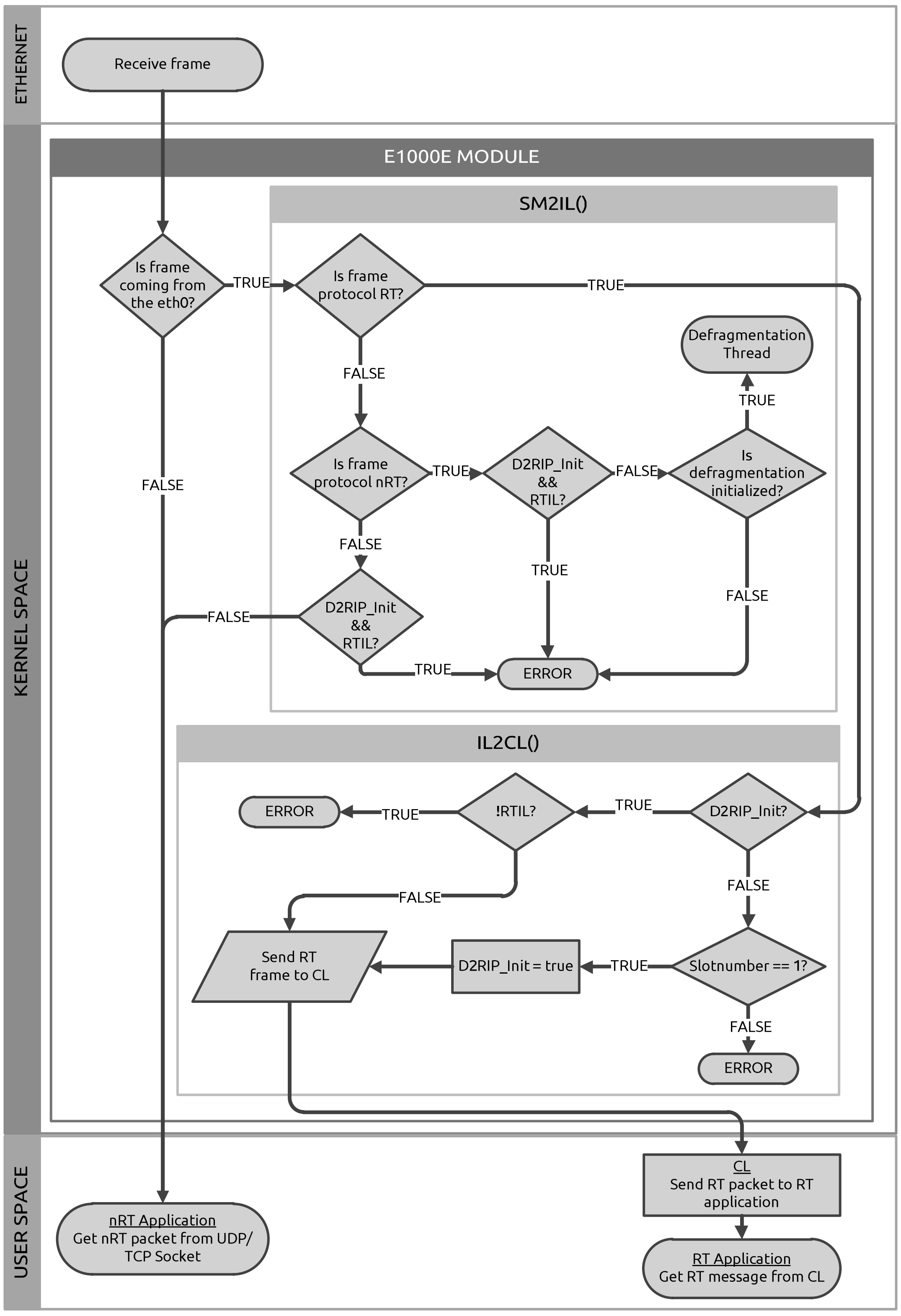


Figure 22 Reception Flowchart

4.9 Changes in D²RIP Protocol

An earlier version of D²RIP was implemented and presented in [14] [13] [15]. Making use of the experiences gained during these studies, in this thesis, the protocol implementation was almost redone. Furthermore, the experimental evaluation was performed with an extended set-up. Many hardware and software changes are made to increase D²RIP performance according to previous version as shown in Table 6 to Table 13. In this section, we compare the current version of D²RIP version with the previous version to show the benefits of the changes.

Table 6 Changes in Hardware

Previous Version	Current Version	Benefits
Each controller used integrated Ethernet controller on the motherboard.	PCI-Express Ethernet card which has a 82574L Ethernet controller is plugged for each controller.	With the hardware time-stamping feature, 82574L Ethernet controller achieves very accurate synchronization between controllers. With the use of an synchronization application that supports hardware time-stamping and 82574L Ethernet controller, time offset between the controllers is decreased the order of 1.5 µs. In the previous implementation of D²RIP, the synchronization between nodes could be done with only software time-stamping. The offset between controllers could increase up to 220 µs. 82574L Ethernet controller has built-in intelligent interrupt management and efficient packet prioritization. Thus, it is capable of sending and receiving frames in lower duration. In addition, the interrupt latencies are improved in multiprocessor systems with MSI-X support.
10 Mbit/s Hub was used for Ethernet communication.	100 Mbit/s Hub is used for Ethernet communication.	Packet transmission speed is increased 10x times. Therefore, the transmission delay is decreased 10x times.

Table 6 (continued)

Previous Version	Current Version	Benefits
Control applications communicated over serial interface (RS-232) with each other.	By adding Ethernet Switch which supports 10/100 Mbit/s speed, control applications communicate over Ethernet.	Serial communication is forcing the system to a particular communication speeds. The communication speed of real-time events message is increased to 100Mbit/s. There was no possibility of sending broadcast messages using the serial channel interface. D²RIP just had the opportunity to point-to-point communication. With the Ethernet interface, broadcast massage sending capability has been added to the D²RIP. The help of Ethernet infrastructure, more nodes can be added in D²RIP according to serial interface.
3 computers (one workcell and two controllers) were used in D²RIP example.	The total number of computers is increased to five **(**one workcell and four controllers).	By using more than two controllers, D²RIP has been started to communicate in the form of broadcast.
D²RIP had been worked only on PCs.	D²RIP has been worked on industrial PC (WAGO) too.	It has been shown that, the new implementation of D²RIP works not only on PCs but also on industrial PCs.
Linux OS was installed on hard-disk drive.	By rearranging the OS, Linux OS is installed on usb flash disk.	D²RIP becomes portable and can be used with devices that have usb-port.

Table 7 Changes in Operating System and Kernel

Previous Version	Current Version	Benefits
D²RIP was used on Ubuntu OS version 10.10.	D²RIP is used on Lubuntu OS version 12.10.	The Linux OS has been changed. Lubuntu OS uses LXDE interface. LXDE interface much faster and lighter than many interfaces in Linux.
Real-time kernel version 2.6.33.7.2-rt30 was used.	Real-time kernel version 3.6.11.1-rt32 is used.	To use hardware time-stamping property with 82574L Ethernet controller properly, the kernel version of Linux OS should be higher than 3.6. Not only do kernel update brings performance increase, but also brings some new functions. With the new version of real-time kernel, real-time performance of Linux OS is increased.
D²RIP was working on the Linux distribution with default settings.	D²RIP works on the Linux distribution which have been performed performance tuning.	A variety of performance settings are made on Linux OS to increase performance and reduce the number of interrupt source. Ethernet card' and HPET clock' interrupt priority are raised.
Limited numbers of settings were changed in the Linux kernel.	Many performance settings are made in the Linux kernel.	By changing many settings in the Linux kernel, IEEE-1588 property is activated and real-time performance is improved.
Many of the settings in the bios were left to the default setting.	Bios settings for the CPU and the power are changed.	Unused devices (audio device, serial and parallel interfaces, etc.) and CPU frequency scaling & power-save mode settings are disabled to reduce interrupt source.

Table 8 Changes in Ethernet Driver and Synchronization Application

Previous Version	**Current Version**	**Benefits**
Ethernet driver version was 1.3.10a for e1000e module.	Ethernet driver version was 2.3.2 for e1000e module.	By upgrading Ethernet driver, hardware-time-stamping property is supported on the driver level.
A lot of code changes were made on the standard Ethernet driver.	The amount of the code that should be inserted into Ethernet driver is reduced.	Many of the D²RIP related codes are moved into IL module. Therefore, the preparation of the other Ethernet drivers for D²RIP is facilitated according to previous D²RIP implementation.
Ethernet driver was loaded with no parameter.	Ethernet driver is loaded with some parameters.	The packet transmission and reception settings of the Ethernet driver are regulated according to D²RIP needs.
Synchronization application supported only software time-stamping.	Synchronization application supports software and hardware-time-stamping.	With the help of hardware time-stamping, there is no need to have guard period in D²RIP. So slot duration is decreased dramatically.
There was no application to synchronize Linux system clock with PTP hardware clock.	An new application is used to synchronize Linux system clock with PTP hardware clock.	Linux system clock synchronizes itself with PTP hardware clock.
There was no modified Ethernet driver for WAGO devices.	e100 Ethernet driver is modified according to D²RIP needs for WAGO devices.	WAGO devices can be used as a controller in D²RIP.

Table 9 Changes in Interface Layer (IL)

Previous Version	Current Version	Benefits
Slot type was divided statically for RT packets and nRT packets.	Slot type was divided dynamically for RT packets and nRT packets.	Slot type is decided dynamically according to D²RIP needs. RT slots become available on the system, if RT communication is needed. Thus, the capacity utilization of the system is improved drastically. nRT slot's right is decided between nodes according to the Round-Robin algorithm.
There was a thread that holds the slot time in IL module. It decided the slot type.	The thread in IL module is removed. All decision code flow is moved into CL application.	Many decision-making algorithms are removed from the kernel space and re-organized in the CL. Thus, D²RIP portability is increased and the integration of the new layer which is developed in the future for D²RIP is facilitated. There is no need to use REQRT action in the protocol. Thus, the time required for this process has been decreased from the slot duration. Threads in the kernel space degrade overall system performance and stability. With this change, overall system performance is increased and usage of system resource is decreased.
There were a lot of unused code snippets and memory allocation in IL module. Furthermore, memory allocation was made dynamically.	Unused code snippets and unnecessary memory allocations are removed. Memory allocations are made statically at the beginning of the system.	Code readability is increased and real-time performance of the IL module is increased. With the statically memory allocation, there is no memory contention.

Table 9 (continued)

Previous Version	Current Version	Benefits
Fragmenter module was waked up with small and larger nRT packets.	Fragmenter module is waked up only with larger nRT packets.	There is no need to wake up fragmenter module for all non-real-time packets. Therefore, fragmenter module uses system resources when the larger nRT packet comes from user-space.
The sendStartData module initiated the system simultaneously in all controllers.	D²RIP starts with SIGUSR1 signal.	The initiation of the D²RIP is improved. Thus there is no need to use a module to start system and D²RIP' resource needs are decreased.
No header information was added in IL.	Packet reception and transmission time are added into packet header in IL.	Performance information is calculated from the time-stamping info.
IL was designed according to point-to-point communication.	IL is designed according to broadcast communication.	IL is re-designed according to multiple controllers' communication. Therefore, D²RIP can support more nodes and send broadcast messages to all nodes.

Table 10 Changes in Coordination Layer (CL)

Previous Version	Current Version	Benefits
There were two threads in CL application.	The number of the thread is dropped to one.	This is the only thread that has highest priority in the Linux OS.
There was a lot of unused code snippets and memory allocation in CL application. Furthermore, memory allocation was made dynamically.	Unused code snippets and unnecessary memory allocations are removed. Memory allocations are made statically at the beginning of the system.	Code readability is increased and real-time performance of the CL application is increased. Dynamic memory access leads to contention and undetermined delays in OS. With the statically memory allocation, there is no memory contention.
Slot-type was pre-determined in D²RIP.	Slot-type decision is made dynamically with using two priority queues.	Utilization of the shared medium is increased significantly.
Standard sleep functions were used in waiting events.	More accurate methods are used to sleep precisely.	Waking up at the desired time is a big challenge in Linux. Therefore, the new spinlock algorithm is developed to achieve accurate sleep-time in D²RIP.
No header information was added in CL.	New header information is added into packet header in CL.	Slot drift and packet size are controlled with the help of CL header information.
D²RIP hadn't got any logging capacity.	C libraries are added into D²RIP to log and measure performance.	The time information of the frames can be easily accessible when D²RIP is running.

Table 10 (continued)

Previous Version	Current Version	Benefits
The RT packets contained null data.	CL application sends only the required RT data to IL.	Unused data is removed from the RT packets. Therefore, communication duration between IL and CL is drooped dramatically.
CL was designed according to point-to-point communication.	CL is designed according to broadcast communication.	CL is re-designed according to multiple controllers' communication. Therefore, D²RIP can support more controllers.
To start up the D²RIP protocol, a lot of pre-settings should be made in OS.	A bash script is written to start system automatically.	D²RIP can be started up without the need of user interaction.

Table 11 Changes in Simfaudes Plug-In

Previous Version	Current Version	Benefits
XML files were prepared according to 2 controllers' example.	XML files were prepared according to 4 controllers' example.	It is shown that D²RIP works with the multiple controllers.
No header information was added in Simfaudes plug-in.	New header information is added into real-time massage in Simfaudes plug-in.	Performance information is calculated from the time-stamping info.

Table 12 Changes in Non Real-Time Application

Previous Version	Current Version	Benefits
There was no nRT traffic application.	An nRT traffic application is developed to measure the delay of nRT packets.	nRT traffics and performance can be measured by using this nRT application.

Table 13 Changes in Real-Time Simulator Application

Previous Version	Current Version	Benefits
There was no RT simulator application.	An RT simulator application is developed to generate more RT traffic.	To increase RT packet traffic, an application is written. Therefore, the limits of D^2RIP can be measured easily.

CHAPTER 5

EXPERIMENTAL EVALUATION OF D^{2}RIP

This chapter presents our experimental evaluation of D^{2}RIP based on the work-cell example in Section 3.7. We first describe the experimental setup in Section 5.1. Then, we measure the OS performance in Section 5.2, we perform a through timing analysis of all protocol components in Section 5.3 and assess the performance of D^{2}RIP for both RT and nRT traffic support in different experimental scenarios in Section 5.4.

5.1 Experimental Setup

We realized the example control application in Section 3.7 where the communication among the controller devices is carried out with our industrial communication protocol D^{2}RIP. To this end, we developed the experiment setup as shown in Figure 23 which consists of the workcell and four distributed controllers each realized with a separate computer. The software and hardware architecture of each controller node is shown in Figure 25. The controller nodes read sensor signals directly from and write actuator signals directly to the workcell on a separate dedicated line that is realized on a separate conventional shared-medium Ethernet network (eth1). Accordingly the simulator on each controller node runs the respective control algorithm which determines the relevant queries/notifications and commands as described in Section 3.7. The simulator then hands out these controller events to the libfaudes D^{2}RIP IO plug-in module. This is a software module that assembles the control application messages and the communication requests as defined in Section 3.3. The software module looks up application-dependent preconfigured requests for each communication event.

Here we would like to note that, in addition to the set-up mentioned above, we ran the example control application in Section 3.7 on the *Wago-I/O-IPC* as shown in Figure 24 Experiment Set-up with WAGO's. We demonstrated the correct operation of D^{2}RIP on Wago devices. However, The Ethernet controller of the *Wago-I/O-IPC* supports only software time stamping in IEEE 1588. Therefore, the synchronization accuracy between the nodes varies too much and we need to add guard periods at the beginning of the every slot to compensate the offset. Consequently, we needed to double the slot duration according to first experiment set-up in Figure 23. In this thesis, we analyze only the D^{2}RIP experiments results achieved on the first experiment set-up in Figure 23.

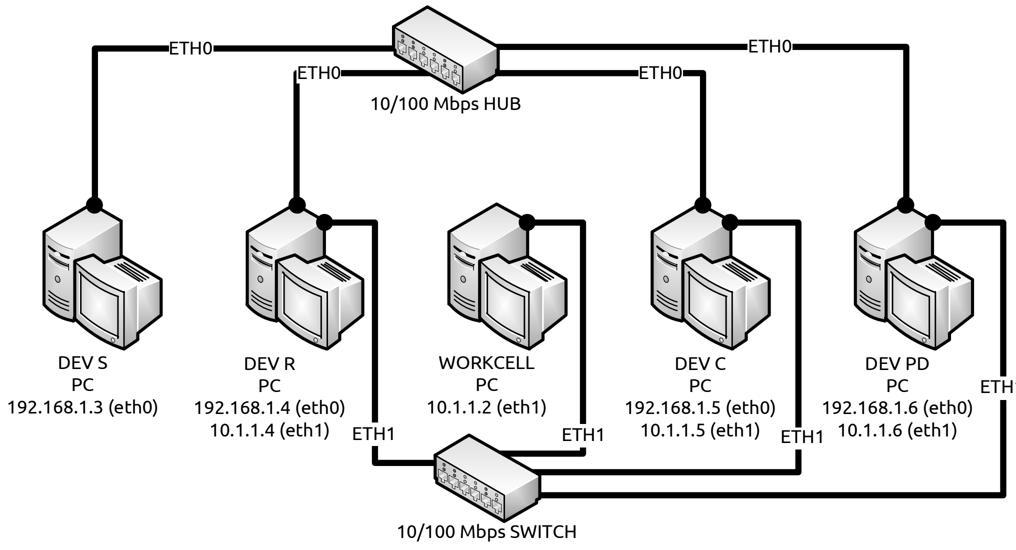


Figure 23 Experiment Set-up

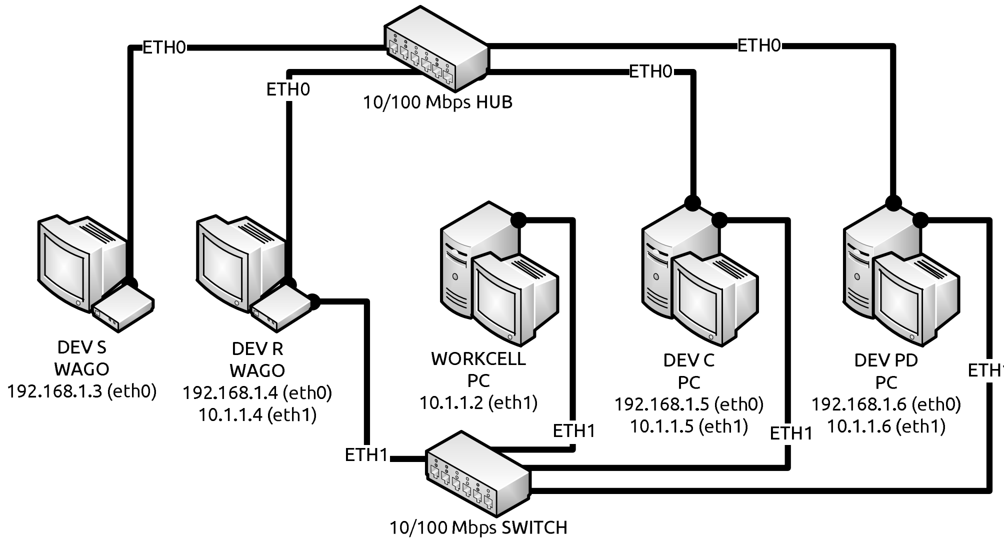


Figure 24 Experiment Set-up with WAGO's

The computers that realize the controller nodes and run D^{2}RIP have

1) PC: QuadCore Intel(R) Core(TM) i3 CPU 550@3.20GHz, 4 GByte RAM, onboard Ethernet controller and IEEE 1588 enabled Ethernet adaptor
2) WAGO: Intel Celeron 600MHz, 256MByte RAM, two onboard Ethernet controllers

The computer that simulates the workcell has a

PC: QuadCore Intel(R) Core(TM) i3 CPU 550@3.20GHz, 4 GByte RAM, onboard Ethernet controller

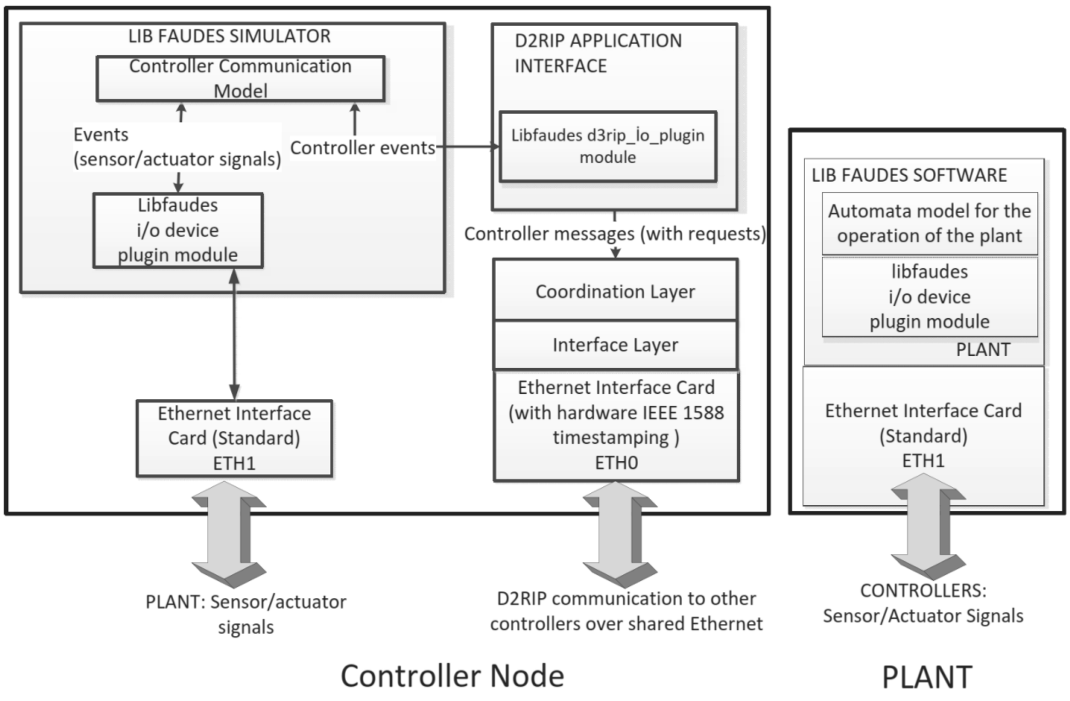

LIB FAUDES SIMULATOR
Controller Communication Model
Events (sensor/actuator signals)
Controller events
Libfaudes i/o device plugin module
D2RIP APPLICATION INTERFACE
Libfaudes d3rip_io_plugin module
Controller messages (with requests)
Coordination Layer
Interface Layer
Ethernet Interface Card (Standard) ETH1
Ethernet Interface Card (with hardware IEEE 1588 timestamping) ETH0
LIB FAUDES SOFTWARE
Automata model for the operation of the plant
libfaudes i/o device plugin module
PLANT
Ethernet Interface Card (Standard) ETH1
PLANT: Sensor/actuator signals
D2RIP communication to other controllers over shared Ethernet
CONTROLLERS: Sensor/Actuator Signals
Controller Node
PLANT

Figure 25 Architecture of the Controller Node and the Plant

5.2 Performance Analysis of RTOS

In this section, we show our modified RTOS performance and the amount of system resources needed by D²RIP. The *iostat* command in Linux generates the CPU utilization report. The first CPU utilization report was taken when the OS booted and the second report was taken after the D²RIP was executed. It can be seen that D²RIP uses only 98.15 – 89.41 = 8.74% of the CPU resources in terms of time-averaged percentages.

avg-cpu:	*%user*	*%nice*	*%system*	*%iowait*	*%steal*	*%idle*
	1.05	*0.00*	*0.42*	*0.38*	*0.00*	*98.15*

avg-cpu:	*%user*	*%nice*	*%system*	*%iowait*	*%steal*	*%idle*
	1.24	*0.00*	*8.23*	*1.12*	*0.00*	*89.41*

The maximum memory usage of the D²RIP can be inspected by the *top* command in Linux. The first memory utilization report was taken when the OS booted and the second report was taken after the D²RIP was executed. The memory difference is 327924 – 321524 = 6.4 MByte after D²RIP was executed.

```
KiB Mem:   3359452 total,   321524 used,  3037928 free,    20428 buffers
```

```
KiB Mem:   3359452 total,   327924 used,  3031528 free,    21680 buffers
```

We next determine the RT capabilities of our RTOS. *Cyclictest* [58] tool that is a high resolution test program is used to measure the worst-case latency of the RTOS. It takes a time snapshot just prior to waiting for a specific time interval (t1), then takes another time snapshot after the timer ends (t2), then comparing the theoretical wakeup time with the actual wakeup time (t2 - (t1 + sleep_time)). This value is the latency for that timer wakeup.

The first test result was taken when OS booted. In this test, we created three threads that have different SCHED_FIFO real-time priorities. The worst-case latency of the thread that has max priority (99) is 60 µs. The worst-case latency of the second thread that has priority 98 is 103 µs. The worst-case latency of the last thread that has min priority (1) is 156 ms. As it seen in the first test, SCHED_FIFO real-time priorities determine the thread's latency significantly. The second test result is taken after the D²RIP was executed. We run *Cyclictest* tool with the max priority (99) to find the worst-case latency for D²RIP protocol. 143 µs is the worst-case latency for our RTOS. This worst-case value is unacceptably large according to our slot duration 250 µs. Therefore, we developed a partially sleeping algorithm that was explained in Section 4.4 to decrease the worst-case latency to values that are less than 30 µs.

```
P:99  I:100  C: 742087  Min: 5  Act:   8   Avg:   12   Max:        60
P:98  I:100  C: 742853  Min: 3  Act: 16   Avg:   22   Max:       103
P: 1   I:100  C: 277245  Min: 4  Act: 14   Avg: 496   Max:  156530
```

```
P:99  I:100  C: 114176  Min: 1  Act:   10   Avg:   13   Max:     143
```

5.3 Timing Analysis of D²RIP

In this section, we present our timing measurements from the hardware setup in the Section 5.1. For each measurement, we obtain a certain number of samples (a number of 5 million slots) and we list the mean, maximum and minimum of the measured values. In addition, we report the 99 % confidence interval. In other words, the average values that we present are sample averages and they stay around a $\pm\Delta$ % neighborhood of the true mean value with a probability of 99%, whereby Δ is denoted as the confidence interval [59].

We note that all timings of D²RIP rely on the precise clock synchronization among the nodes that is achieved by IEEE 1588 as described in Sections 2.2.3 and 4.2. Hence, we first consider our clock synchronization measurements as listed in Table 14. Here, the *accuracy* is the difference among the master and slave clocks [1].

We run our synchronization application with hardware time stamping feature in our first setup showed in Figure 23. It can be seen that very small values of accuracy in the order of µs are achieved. This is in compliance with results for IEEE 1588 synchronization with hardware time stamping in the literature [60]. Next, we run the synchronization application with software time stamping feature in our second setup showed in Figure 24. *Wago-I/O-IPCs* support only software time stamping in IEEE 1588. It is seen that the synchronization performance with software timestamping is not as promising as the previous experiment. The obtained results indicate that, in principle, time slotted access to the shared medium can be implemented with a precision in the order of µs using hardware time stamping.

Table 14 IEEE 1588 Accuracy Results

	Mean µs	**%± Confidence Interval** (99% confidence)	**Max.** µs	**Min.** µs
Hardware Accuracy (µs)	1.4	%3.71	4.2	0
Software Accuracy (µs)	105.3	%5.93	204,3	9.4

We next evaluate the further delay components that contribute to the execution of each time slot according to Figure 2. In addition to each of the listed actions, we note that the periodic wake-ups of the CL thread that is triggered by CLOCK_REALTIME introduce additional delays (see Section 4.4). Hence this wake-up time is also listed since it constitutes an important component of the time elapsed for the actions executed in a time slot. The evaluation is done for 5 million slots from all nodes of the hardware setup described in Section 5.1. The time slot size in the experiment is chosen as 250 µs and massage length is chosen as 504 Bytes. Each eligibility time is eT = 4 ms for all events. Therefore, all the actions executed in 16 times slot and we take the averages over 312500 samples. According to the result in Table 15, it remains 60 µs to transmit message on 100 Mbit/s Ethernet and delays for Hub.

Table 15 Timings of Actions Taken in an RT Slot

	Mean µs	**%± Confidence Interval** (99% confidence)	**Max.** µs	**Min.** µs
CL thread wake-up	0.7	%0.21	30.5	0.7
AP2CL	3.2	%3.23	12.1	2.3
CLUPDATE	1.3	%0.45	10.5	0.7
IL2SM	12.2	%0.38	27.8	9.8
SM2IL	21.4	%0.38	48.0	9.8
IL2CL	9.2	%1.22	46.1	1.0
CL2AP	6.1	%4.07	16.7	4.9
Completion time in slot	94.1	%1.82	231.1	69.2

It is readily observed that, on average, the largest delays are introduced by the actions IL2SM and SM2IL which require data exchange between the kernel space and Ethernet line. In total, all actions to be completed in each time slot add up to an average execution delay of 54.1 µs. Adding the message transmission delay of 40 µs for 504 Byte messages, the average completion time for each slot is 94.1 µs as is shown in the Table 15. However, it turns out that large maximum delays are observed for the CL thread wake-up and message transmission between the user space and kernel space. Since the execution of each time slot has to be correct, these maximum delays have to be taken into account. As is shown in Table 15, a maximum completion time for each slot of 231.1 µs is found in our experiments. Considering that the slot time was chosen as 250 µs, all time slots execute correctly.

The measurements in Table 15 allow computing the effective RT bandwidth that can be obtained for D²RIP according to (3.1) in Section 3.6. Considering the message length of *L* = 504 Bytes and the slot size of *dSlot* = 250 µs, we obtain

$$B_{eff} = \frac{504 \times 8\,Byte}{250\ \mu\mathrm{s}} = 16.1\ Mbit/s \tag{5.1}$$

It can be seen in Table 1 that the effective RT bandwidth that is achieved for D²RIP is similar to comparable protocols such as EPA, EPL or Tcnet with the difference that D²RIP enables dynamic allocation of RT bandwidth. The RT bandwidth gain can be calculated according to (3.5) in Section 3.6 by using (5.1) in (3.10) and (3.11) in Section 3.7.

$$G_{RT} = \frac{B_{sum}}{B_{max}} = \frac{3.2}{1} = 3.2 \tag{5.2}$$

Considering the RT bandwidth gain of *GRT* = 3.2, the RT bandwidth that is comparable to static protocols for our application example is

$$B_{eff}B_{RT} = 16.1\ Mbit/s \times 3.2 = 51.52\ Mbit/s \tag{5.3}$$

5.4 Performance Experiments and Results

We next demonstrate the operation and the performance of our D²RIP stack with the example case study. To this end, we perform three experiments that are targeted to show the delivery times, nRT properties and dynamic RT allocation properties of D²RIP. To this end, experiment 1, provides detailed delay measurements for the RT traffic of the workcell example. In experiment 2, we add another RT application to dynamically increase and decrease the RT traffic. Finally, in experiment 3, we investigate the throughput and delay values for nRT traffic under different scenarios.

5.4.1 Experiment 1: Detailed investigation for RT traffic service of D^2RIP

In this experiment, the control application that is described in Section 3.7 is run on D^2RIP for a slot time of 250 µs and an RT message length of 504 Bytes. We first verified that all controller events were communicated successfully and the workcell operation proceeds as specified. Next, we considered the end-to-end delays of message transmissions between the different software layers IL, CL and AL. To this end, we measured the respective delays for all message transmissions of our RT control application with a number of 5 million slots as listed in Table 16.

Table 16 End-to-end Delays of RT Messages of the Workcell Example

	Mean µs	**%± Confidence Interval** (99% confidence)	**Max.** µs	**Min.** µs
IL to IL	80.2	%6	108	72
CL to CL	90.1	%5.60	142	56
AL to AL	3473.1	%1.96	4423	221

The measurements show that the end-to-end delays between the ILs and CLs of different nodes are bounded by a maximum value of 108 µs and 142 µs, respectively. Such value is expected according to the accumulated delay components between the actions IL2SM-SM2IL and CLUPDATE–IL2CL according to Table 15. In particular, these end-to-end delays remain within the slot time of 250 µs which is a prerequisite for the correct time slot execution. The end-to-end delay between the application layers (AL to AL) constitutes the delivery time of RT messages [1]. Its minimal value of 221 µs represents the case where the application generates a message right at the moment where the transmitting node obtains a transmission slot. In this case, the ready message is polled by the CL and reaches the AL of the receiver nodes within this transmission slot. Accordingly, the minimum delivery time is smaller than the slot time of 250 µs. On the other hand, the maximum delivery time represents the case where an application message is generated in a transmission slot of the transmitter node but after the message should have been polled by the CL. In that case, this node has to wait until its next request becomes eligible, which takes one slot time plus the eligibility time of 4 ms. Then, the message will be transmitted within the subsequent transmission slot which is bounded by 250 µs. Together, we expect a maximum delay of 4.5 ms which complies with the measurement. Note that the same result is given by the worst-case delivery time according to (3.3) in Section 3.6 considering the maximum number of requests in the priority queue was determined as 1:

$$w_{req} = dSlot \times (Q_{req} + 1) + eT_{req} = 250\mu s \times 2 + 4ms = 4.5ms \tag{5.4}$$

5.4.2 Experiment 2: Varying RT Traffic Input

In this experiment, we considered the RT traffic that is generated when running the workcell example setup, whereby only one node can have an eligible communication request at a time. In this experiment we generate additional RT background traffic that is generated on channel 2 of the controller nodes PLC-S and PLC-R. This RT traffic is designed such that channel 2 of PLC-S transmits messages to PLC-R with a deadline of 5 ms and an eligibility time of 2 ms and puts a request for message transmission of channel 2 of PLC-R with deadline 2 ms and eligibility time 1 ms. Whenever channel 2 of PLC-R transmits, it puts back a request for PLC-S with deadline 5 ms and eligibility time 2 ms. Accordingly, the maximum number of requests in the priority queue is now $Q_{req} = 2$ and the worst-case delivery times for RT messages evaluate to

$$w_1 = 250\mu\text{s} \times 3 + 4\text{ms} = 4.75ms \tag{5.5}$$

$$w_2 = 250\mu\text{s} \times 3 + 2\text{ms} = 2.75ms \tag{5.6}$$

$$w_3 = 250\mu\text{s} \times 3 + 1\text{ms} = 1.75ms \tag{5.7}$$

Here, w_1 denotes the worst-case delivery time for RT messages of the workcell example, w_2 denotes the worst-case delivery time of the RT messages from channel 2 of PLC-S (slow mode) and w_3 denotes the worst-case delivery time of RT messages from channel 2 of PLC-R (fast mode). The following Table 17 shows the measurement results from our experimental setup. Again, a number of 5 million slots are evaluated.

Table 17 Delivery-time Measurements with Additional RT Traffic

	Mean µs	**%± Confidence Interval** (99% confidence)	**Max.** µs	**Min.** µs
AL to AL (workcell)	3542.1	%2	4745	211
AL to AL (slow mode)	1800	%1.9	2534	230
AL to AL (fast mode)	800	%2.3	1620	225

It is readily observed that the measured worst-case delivery times are bounded by the previously computed values. Hence, it is confirmed that each message on D²RIP will meet its specified deadline if the relevant parameters such as time slot and eligibility time are chosen properly. Furthermore, this experiment validates the case of multiple nodes competing simultaneously for medium access. Our slot allocation scheme based on communication requests successfully gives medium access to the node with the most urgent eligible request such that no deadlines are violated.

5.4.3 Experiment 3: Support for nRT Traffic in D²RIP

We finally investigate the support of nRT traffic by D²RIP. According to the previous sections, it is first possible to determine the maximum amount of bandwidth required for the RT messages generated by the workcell example

$$B_{max} = \frac{dSlot}{4\,ms} \times B_{eff} = \frac{250\mu s}{4\,ms} \times 16.1\,Mbit = 1\,Mbit/s \tag{5.8}$$

Hence, an nRT bandwidth of

$$B_{nRT} = B_{eff} - B_{max} = 16.1\,Mbit/s - 1\,Mbit/s = 15.1\,Mbit/s \tag{5.9}$$

is left for nRT traffic, whereby it has to be considered that long messages are fragmented in order to comply with the maximum message size of 504 Bytes. Next, we investigate the delivery times of nRT packets between application layers under different nRT traffic loads, keeping the RT traffic the same as in Experiment 1. We collected measurements under four average nRT traffic patterns where a message is generated every 500 µs, 1 ms, 10 ms and 100 ms, respectively by each node. Using these traffic patterns, we consider the case of short messages (40 Bytes), medium size messages (576 Bytes) and long messages (1500 Bytes). Our measurement results are shown in Table 18, Table 19 and Table 20. Note that the last column shows the consumed bandwidth by nRT traffic for the respective traffic pattern.

Table 18 nRT Traffic with 40 Byte Message Length

Transmission Period	**Mean** µs	**%± Confidence Interval** (99% confidence)	**Max.** µs	**Min.** µs	**Bandwidth** Mbit/s
500 µs	——	——	——	——	2.56
1 ms	1530.1	%0.83	5940	90	1.28
10 ms	636.2	%2.03	3295	57	0.128
100 ms	620.1	%6.92	3124	64	0.01228

Table 19 nRT Traffic with 576 Byte Message Length

Transmission Period	Mean µs	%± Confidence Interval (99% confidence)	Max. µs	Min. µs	Bandwidth Mbit/s
2 ms	——	——	——	——	9.2
2.5 ms	2135	%0.47	10235	1756	7.4
5 ms	1728.4	%0.59	5354	1081	3.7
10 ms	1683.2	%0.82	4320	1082	1.85
100 ms	1696.3	%2.65	3853	1079	0.185

Our first observation from these measurements is that the supported nRT bandwidth depends on the traffic pattern. In particular, Table 18 shows that very small nRT packets lead to a low possible nRT bandwidth. This is expected since a supported packet length of 504 Bytes is chosen for the conducted experiment. Hence, only a very small fraction of the time slot is used by short nRT packets. Second, we conclude that a nRT bandwidth that is close to the computed maximum of 15.1 Mbit/s can be achieved for long packets even if fragmentation is needed. Realistic nRT message delivery times are obtained for packet sizes of 576 Bytes and 1500 Bytes at nRT bandwidths of 7.4 Mbit/s and 9.6 Mbit/s, respectively. Here we would like to note that synchronization packets are dropped the nRT bandwidth because of high priority.

Table 20 nRT Traffic with 1500 Byte Message Length

Transmission Period	Mean µs	%± Confidence Interval (99% confidence)	Max. µs	Min. µs	Bandwidth Mbit/s
2.5 ms	——	——	——	——	19.2
5 ms	3842.3	%0.28	6698	3101	9.6
10 ms	3844.3	%0.38	6541	3079	4.8
100 ms	3827.5	%1.26	6118	3087	0.48

CHAPTER 6

CONCLUSIONS AND FUTURE WORKS

The subject of this thesis is the implementation of the novel industrial Ethernet protocol D^{2}RIP with real-time (RT) and non-real-time (nRT) traffic support. D^{2}RIP is a fully distributed communication protocol with time-slotted medium access, whereby the ownership of each time slot is decided instantaneously based on application specific data. That is, different from other industrial Ethernet protocols, D^{2}RIP is particularly efficient for applications, whose bandwidth requirements change dynamically.

Regarding the implementation of D^{2}RIP, we first define appropriate data structures for the protocol implementation such that D^{2}RIP can run on top of COTS Ethernet hardware. Then, we realize time-slotted medium access with the help of precise clock synchronization using the IEEE 1588 synchronization protocol. Our obtained time slot size on 100 Mbit/s Ethernet is 250 µs. Based on a distributed computation on each network node, each time slot is dynamically allocated to a unique node that can send RT messages up to a size of 504 Byte or nRT messages. We provide a packet fragmentation thread in order to fit long nRT packets into the available time slots. The protocol implementation is accompanied by detailed measurements on an experimental hardware setup with a realistic industrial application example adapted from a manufacturing system example in [61] and [62]. As the main results, D^{2}RIP supports an effective RT bandwidth of 16.1 Mbit/s which is similar to comparable protocols. However, D^{2}RIP uses this effective bandwidth more efficiently due to the dynamic bandwidth allocation. In addition, we obtain clock synchronization accuracies in the order of 1.5 µs and support delivery times below 5 ms. Regarding nRT traffic, we observe that short packets are not efficiently supported by D^{2}RIP due to the time-slotted medium access.

One of the our contribution is implementing the two layer D^{2}RIP stack on COTS Ethernet hardware using the open source OS Linux with RT-Preempt kernel. We chose Lubuntu that is a lightweight variant of Ubuntu as our OS because it is a fast and lightweight OS. Lubuntu OS is intended to have low-resource system requirements and is designed primarily for mobile devices and embedded controllers. Standard Linux kernel in Lubuntu OS is not suitable for real-time operation because it doesn't support hard real-time scheduling. Therefore we patched standard Linux kernel with RT-Preempt kernel to add hard real-time capabilities to a Lubuntu OS to facilitate the development of D^{2}RIP.

The RT bandwidth utilization of D^{2}RIP is improved dramatically according to the previous implement of D^{2}RIP. In the previous implementation, time slot size was 3 ms and a node could send messages up to a size of 150 Byte. Thus, earlier D^{2}RIP supports an effective RT bandwidth of 0.4 Mbit/s. As a result, the efficiency of Ethernet bandwidth usage is increased **40x times** according to earlier implementation.

The implementation of D²RIP is based on error-free communication. D²RIP assumes that no faults occur during its operation. In case of RT packet loss, the operation of D²RIP will get stuck. Therefore, a new layer plane that we call the *Dependability Plane* (DP) with interfaces to the application layer, Interface Layer, Coordination Layer and the shared medium broadcast network is currently under development as a part of a Ph.D. thesis work [63]. There is a study on dependability plane that realizes a new distributed rollback method with synchronized check pointing strategy. DP is connected to our layers CL and IL. A dependability header is piggybacked on messages and local copies of the state variables are stored with synchronized check pointing strategy. An acceptance test is occurred at the end of each transmission slot. A rollback message is transmitted in the first transmission slot of a node that is certain about a fault or failure occurrence. There are some conditions to make DP work successfully. There should be at least 3 nodes in the network and they should obtain a transmission slot regularly. Moreover, there should be no faults at the beginning of the network operation and no fault should be happened before the previous fault situation is resolved.

The preliminary version of DP is implemented and integrated to the D²RIP implementation in this thesis [64].

Although D²RIP is implemented for shared-medium Ethernet, its operation is suitable for wireless protocols such as Wireless Hart (IEEE 802.11), which is a possible direction for future work.

REFERENCES

[1] M. Felser, Real-time Ethernet for automation applications, in: R. Zurawski (Ed.), Embedded Systems Handbook, Second Edition: Networked Embeddedded Systems, CRC Press, Inc., Boca Raton, FL, USA, 2009, pp. 21–1–21–20, 2nd edition.

[2] T. Sauter and A. Treytl, Communication systems as an integral part of distributed automation systems, in: H. Kühnle (Ed.), Distributed Manufacturing, Springer London, 2010, pp. 93–111.

[3] J. Moyne and D. Tilbury, The emergence of industrial control networks for manufacturing control, diagnostics, and safety data, Proceedings of the IEEE 95 (1) (2007) 29–47.

[4] P. Neumann, Communication in industrial automation - what is going on?, Control Engineering Practice 15 (2007) 1332–1347.

[5] L. Seno, F. Tramarin, and S. Vitturi, Performance of industrial communication systems: Real application contexts, Industrial Electronics Magazine, IEEE 6 (2) (2012) 27–37.

[6] J.-D. Decotignie, The many faces of industrial Ethernet [past and present], Industrial Electronics Magazine, IEEE 3 (1) (2009) 8–19.

[7] P. Gaj, J. Jasperneite, and M. Felser, Computer communication within industrial distributed environment — a survey, Industrial Informatics, IEEE Transactions on 9 (1) (2013) 182–189.

[8] K. C. Lee and S. Lee, Performance evaluation of switched Ethernet for realtime, Computer Standards and Interfaces 24 (5) (2002) 411–423.

[9] P. Pedreiras, P. Gai, L. Almeida, and G. Buttazzo, FTT-Ethernet: a flexible real-time communication protocol that supports dynamic QoS management on Ethernet-based systems, Industrial Informatics, IEEE Transactions on 1 (3) (2005) 162–172.

[10] IEEE 1588 standard, http://ieee1588.nist.gov/, last visited on August 2013.

[11] D. E. Kaynar, N. Lynch, R. Segala, and F. Vaandrager, The theory of timed I/O, Tech. Rep. MIT-LCS-TR-917, MIT Laboratory for Computer Science, Cambridge, MA (2003).

[12] K. Schmidt and E. Schmidt, Distributed real-time protocols for industrial control systems: Framework and examples, Parallel and Distributed Systems, IEEE Transactions on 23 (10) (2012) 1856–1866.

[13] Ahmet Korhan Gözcü, Implementation and evaluation of a eynchronous time-slotted medium access protocol for networked industrial embedded systems, M.S. Thesis, Defense date: September 2011.

[14] Ulaş Turan, Implementing and evaluating the Coordination Layer and time-synchronization of a new protocol for industrial communication networks, M.S. Thesis, Defense date: September 2011.

[15] A. K. Gözcü, U. Turan, E. G. Schmidt, and K. Schmidt, SIU Paper: Dinamik Dağitik Gerçek Zamanli Endüstriyel İletişim Protokolu (D2GEP) Gerçekleştirimi-The implementation of Dynamic Distributed Real time Industrial communication Protocol (D2RIP), IEEE 20. Sinyal İşleme ve İletişim Uygulamaları Kurultayı, 2012.

[16] S. Vitturi, On the use of Ethernet at low level of factory communication systems, Computer Standards and Interfaces 23 (4) (2001) 267–277.

[17] D. Nesic, A. Teel, and D. Carnevale, Explicit computation of the sampling period in emulation of controllers for nonlinear sampled-data systems, Automatic Control, IEEE Transactions on 54 (3) (2009) 619–624.

[18] K. Schmidt, E. Schmidt, and J. Zaddach, A shared-medium communication architecture for distributed discrete event systems, Control & Automation, Mediterranean Conference on (2007) 1856–1866.

[19] K. Schmidt, E. Schmidt, and J. Zaddach, Safe operation of distributed discrete event controllers: A networked implementation with real-time guarantees, in: IFAC World Congress, 2008, pp. 4126–4131.

[20] J.-D. Decotignie, Ethernet-based real-time and industrial communications, Proceedings of the IEEE 93 (6) (2005) 1102–1117.

[21] E. Schemm, Sercos to link with Ethernet for its third generation, Computing & Control Engineering Journal 15 (2) (2004) 30–33.

[22] Real-time Ethernet: SERCOS III: Proposal for a publicly available specification for real-time Ethernet, Doc. IEC 65C/358/NP (2004).

[23] Real-time Ethernet: Profinet IO: Proposal for a publicly available specification for real-time Ethernet, Doc. IEC 65C/359/NP (2004).

[24] D. Jansen and H. Buttner, Real-time Ethernet the EtherCAT solution, Computing & Control Engineering Journal 15 (1) (2004) 16–21.

[25] Real-time Ethernet: Ethernet control automation technology EtherCAT: Proposal for a publicly available specification for real-time Ethernet, Doc. IEC 65C/355/NP (2004).

[26] Schneider automation – modbus messaging on TCP/IP implementation guide, http://www.modbus.org/, last visited on August 2013.

[27] S.-K. Kweon and K. Shin, Statistical real-time communication over Ethernet, Parallel and Distributed Systems, IEEE Transactions on 14 (3) (2003) 322–335.

[28] Ethernet/IP library, http://www.odva.org/, last visited on August 2013.

[29] Real-time Ethernet: Ethernet/IP with time synchronization: Proposal for a publicly available specification for real-time Ethernet, Doc. IEC 65C/361/NP (2004).

[30] L. Liu and G. Frey, Simulation approach for evaluating response times in networked automation systems, Emerging Technologies & Factory Automation, IEEE Conference on (2007) 1061–1068.

[31] J.-L. Scharbarg and C. Fraboul, Simulation for end-to-end delays distribution on a switched Ethernet, Emerging Technologies & Factory Automation, IEEE Conference on (2007) 1092–1099.

[32] X. Fan, M. Jonsson, and J. Jonsson, Guaranteed real-time communication in packet-switched networks with FCFS queuing, Comput. Netw. 53 (3) (2009) 400–417.

[33] K. Schmidt and E. Schmidt, A longest-path problem for evaluating the worst-case packet delay of switched Ethernet, in: Industrial Embedded Systems, International Symposium on, 2010, pp. 205–208.

[34] R. Moraes et al., Enforcing the timing behavior of real-time stations in legacy bus-based industrial Ethernet networks, Computer Standards and Interfaces 33 (3) (2011) 249–261.

[35] G. Cena, L. Seno, A. Valenzano, and S. Vitturi, Performance analysis of Ethernet powerlink networks for distributed control and automation systems, Computer Standards and Interfaces 31 (3) (2009) 566–572.

[36] Real-time Ethernet: EPA (Ethernet for plant automation): Proposal for a publicly available specification for real-time Ethernet, Doc. IEC 65C/357/NP (2004).

[37] Z. Wei, X. Aidong, and S. Yan, Theory and implementation of real-time testing in EPA, in: Mechatronics and Automation, Inter. Conference on, 2010, pp. 778–782.

[38] Real-time Ethernet: EPL (Ethernet powerlink): Proposal for a publicly available specification for real-time Ethernet, Doc. IEC 65C/356a/NP (2004).

[39] S. Vitturi, L. Peretti, L. Seno, M. Zigliotto, and C. Zunino, Real-time Ethernet networks for motion control, Computer Standards and Interfaces 33 (5) (2011) 465–476.

[40] Real-time Ethernet: TCnet (Time-Critical Control Network): Proposal for a publicly available specification for real-time Ethernet, Doc. IEC 65C/353/NP (2004).

[41] J. C. Eidson, Measurement, Control, and Communication Using IEEE 1588, Springer, 2006.

[42] Real-Time Linux, http://en.wikipedia.org/wiki/RTLinux/, last visited on August 2013.

[43] Open Source, http://wikipedia.org/wiki/Free_and_open_source_software/, last visited on August 2013.

[44] Lubuntu, http://lubuntu.net/, last visited on August 2013.

[45] Usb Image Tool, http://www.alexpage.de/usb-image-tool/, last visited on August 2013.

[46] The Linux Kernel Archives, http://www.kernel.org/, last visited on August 2013.

[47] K. Erwinski, M. Paprocki, L. Grzesiak, K. Karwowski, and A. Wawrzak, Application of Ethernet Powerlink for communication in a Linux RTAI open CNC system, Industrial Electronics, IEEE Transactions on 60 (2) (2013) 628–636.

[48] Intel e1000e Driver ReadMe, http://downloadmirror.intel.com/22603/eng/README.txt, last visited on August 2013.

[49] Time keeping in Virtual Machines, http://www.vmware.com/files/pdf/techpaper/Timekeeping-In-VirtualMachines.pdf, last visited on August 2013.

[50] D. W. Allan, Precision oscillators: Dependence of frequency on temperature, humidity and pressure, in Proceedings of the 1992 IEEE Frequency Control Symposium, 1992, report of Working Group 3 of the IEEE SCC27 Committee.

[51] R. Cochran and C. Marinescu, Design and implementation of a ptp clock infrastructure for the linux kernel, in Precision Clock Synchronization for Measurement Control and Communication (ISPCS), 2010 International IEEE Symposium on, Sep. 27–Oct. 1, 2010, pp. 116 –121.

[52] R. Cochran, C. Marinescu, and C. Riesch, Synchronizing the Linux System Time to a PTP Hardware Clock.

[53] The Linux PTP project, http://linuxptp.sourceforge.net/, last visited on August 2013.

[54] Intel 82574l Gigabit Ethernet Controller, http://ark.intel.com/products/32209/Intel-82574L-Gigabit-Ethernet-Controller/, last visited on August 2013.

[55] Intel 82574l Gigabit Ethernet Controller driver, http://sourceforge.net/projects/e1000/, last visited on August 2013.

[56] Ethtool, https://www.kernel.org/pub/software/network/ethtool/, last visited on August 2013.

[57] libFAUDES, http://www.rt.eei.uni-erlangen.de/FGdes/faudes/, last visited on August 2013.

[58] Cyclictest, https://rt.wiki.kernel.org/index.php/Cyclictest/, last visited on August 2013.

[59] Semih Bilgen, Confidence of Simulation Results, Lecture Note, METU.

[60] P. Loschmidt, R. Exel, A. Nagy, and G. Gaderer, Limits of synchronization accuracy using hardware support in IEEE 1588, Precision Clock Synchronization for Measurement, Control and Communication, IEEE International Symposium on, 2008, pp. 12–16.

[61] M.H. de Queiroz, J.E.R. Cury, and W.M. Wonham, Multitasking Supervisory Control of Discrete-Event Systems, J. Discrete Event Dynamic Systems: Theory and Applications, vol. 15, pp. 375-395.

[62] K. Schmidt, M. de Queiroz, and J. Cury, Hierarchical and Decentralized Multitasking Control of Discrete Event Systems, Proc. IEEE 46th Conf. Decision and Control, pp. 5936-5941, Dec. 2007.

[63] Y.B. Kartal, Dependable Framework Design for Distributed Real-Time Network Protocols Running On Shared Medium: Design, Simulation and Verification, Phd. Thesis, METU Sept. 2013, Under Preparation.

[64] Ö. B. Sezer, Implementation and Evaluation of the Dependability Plane for the Dynamic Distributed Dependable Real Time Industrial Protocol (D^{3}RIP), M.S. Thesis, Defense date: September 2013.