IMPLEMENTING AND EVALUATING THE COORDINATION LAYER AND TIME-SYNCHRONIZATION OF A NEW PROTOCOL FOR INDUSTRIAL COMMUNICATION NETWORKS

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

BY

ULAŞ TURAN

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

SEPTEMBER 2011

Approval of the thesis:

IMPLEMENTING AND EVALUATING THE COORDINATION LAYER AND TIME-SYNCHRONIZATION OF A NEW PROTOCOL FOR INDUSTRIAL COMMUNICATION NETWORKS

submitted by **ULAŞ TURAN** in partial fulfillment of the requirements for the degree of **Master of Science in Electrical and Electronics Engineering Department, Middle East Technical University** by,

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

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

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

Examining Committee Members:

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

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

Prof. Dr. Gözde Bozdağı Akar
Electrical and Electronics Engineering Dept., METU _____

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

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

Date: _____

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: ULAŞ TURAN

Signature :

ABSTRACT

IMPLEMENTING AND EVALUATING THE COORDINATION LAYER AND TIME-SYNCHRONIZATION OF A NEW PROTOCOL FOR INDUSTRIAL COMMUNICATION NETWORKS

Turan, Ulaş

M.Sc., Department of Electrical and Electronics Engineering

Supervisor : Asst. Prof. Dr. Şenan Ece Schmidt

September 2011, 78 pages

Currently automation components of large-scale industrial systems are realized with distributed controller devices that use local sensor/actuator events and exchange shared events with communication networks. Fast paced improvement of Ethernet provoked its usage in industrial communication networks. The incompatibility of standard Ethernet protocol with the real-time requirements encouraged industry and academic researchers to provide a resolution for this problem. However, the existing solutions in the literature suggest a static bandwidth allocation for each controller device which usually leads to an inefficient bandwidth use. Dynamic Distributed Dependable Real-time Industrial Communication Protocol (D^3RIP) family dynamically updates the necessary bandwidth allocation according to the messages generated by the control application. D^3RIP is composed of two protocols; *interface layer* that provides time-slotted access to the shared medium based on an accurate clock synchronization of the distributed controller devices and *coordination layer* that decides the ownership of real-time slots. In this thesis, coordination layer protocol of D^3RIP family and the IEEE 1588 time synchronization protocol is implemented and tested on the real hardware system that resem-

bles a factory plant floor. In the end, we constructed a system that runs an instance of D^3RIP family with 3ms time-slots that guarantees 6.6ms latency for the real-time packets of control application. The results proved that our implementation may be used in distributed controller realizations and encouraged us to further improve the timing constraints.

Keywords: Industrial communication, real-time, distributed computation, Ethernet

ÖZ

ENDÜSTRİYEL HABERLEŞME AĞLARI İÇİN GELİŞTİRİLEN PROTOKOLÜN KOORDİNASYON KATMANININ VE EŞZAMANLAMASININ GERÇEKLENMESİ VE DEĞERLENDİRİLMESİ

Turan, Ulaş
Yüksek Lisans, Elektrik ve Elektronik Mühendisliği Bölümü
Tez Yöneticisi : Yrd. Doç. Dr. Şenan Ece Schmidt

Eylül 2011, 78 sayfa

Günümüzdeki büyük ölçekli endüstriyel sistemlerdeki otomasyon bileşenlerinde bulunan dağıtılmış kontrolcular algılayıcı/eyleyici gibi yerel sinyallerin yanı sıra haberleşme ağları üzerinden geçen ortak sinyaller de içermektedirler. Hızla gelişen Ethernet protokolü endüstriyel haberleşme ağlarında da kullanılmaktadır. Standart Ethernet prokolünün gerçek zamanlı haberleşmeye olanak sağlamadığından çeşitli önerilerle bu sorun aşılmaya çalışılmıştır. Bu konudaki mevcut öneriler kontrol uygulamarının ağ kapasite kullanımlarını gerçek dışı varsayımlara dayandırmaktadır. Dinamik Dağıtılmış Güvenilir Gerçek Zamanlı Endüstriyel İletişim Protokolü (D^3RIP) ailesi ağ kapasitesi tahsisini kontrol uygulamalarından gelen mesajlarla dinamik olarak güncellemektedir. D^3RIP iki protokolden oluşmaktadır: *arayüz katmanı* paylaşılan ortama dağıtılmış kontrolcuların eşzamanlanmasını temel alarak zaman-dilimli erişimi sağlarken *koordinasyon katmanı* gerçek zamanlı dilimlerin sistemdeki sahibini belirlemektedir. Bu tezde, D^3RIP ailesine ait *koordinasyon katmanı* ve IEEE 1588 eşzamanlama protokolü gerçekleştirilmiş ve gerçek donanımlarla test edilmiştir. Son olarak oluşturduğmuz sistem 3ms zaman-dilimleriyle çalışıp gerçek zamanlı paketler için 6.6ms altında gecikmeyi garanti etmiştir. Bu sonuçlar yaptığımız

gerçekleştirmenin dağıtılmış kontrol sistemleri için kullanılanılabileceğini göstermiştir.

Anahtar Kelimeler: Endüstriyel haberleşme, gerçek zamanlılık, dağıtılmış hesaplama, Ethernet

To My Parents

ACKNOWLEDGEMENTS

I would like to express my special thanks to my supervisor Asst. Prof. Dr. Şenan Ece Schmidt who gave me the opportunity to do this thesis on the topic Industrial Communication Networks. I am heartily thankful to Klaus Schmidt for his assistance throughout my work. My special thanks goes to TÜBITAK for financing the project and providing me the chance to visit University of Erlangen in Germany. It was a very kind of Prof. Thomas Moor from University of Erlangen to invite me to the Control Department and assisting me to complete the integration process. My colleague Ahmet Korhan Gözcü had great support for my thesis study from the beginning, I am thankful for him. I would like to special reference for my employer, ASELSAN for encouraging me to complete my studies in the university. Finally, I would also like to thank my parents and friends who helped me a lot in finishing this thesis work within the limited time.

TABLE OF CONTENTS

LIST OF TABLES

TABLES

LIST OF FIGURES

FIGURES

LIST OF ABBREVIATIONS AND ACRONYMS

ADO	Allocation Data Object
API	Application Programming Interface
CL	Coordination Layer
CRC	Cyclic Redundancy Check
CSMA/CD	Carrier sense multiple access with collision detection
D^3RIP	Dynamic Distributed Dependable Real-time Industrial communication Protocol family
DART	Dynamic Allocation Real-time Protocol
DES	Discrete Event System
FPGA	Field Programmable Gate Array
HPET	High Precision Event Timer
IL	Interface Layer
IPC	Industrial PC
IRQ	Interrupt Request
ISR	Interrupt Service Routine
MAC	Medium Access Control

nRT	non-Real Time
NIC	Network Interface Controller
NTP	Network Time Protocol
OSADL	Open Source Automation Development Labs
OSI	Open System Interconnection
PLC	Programmable Logic Controller
POSIX	Portable Operating System Interface for Unix
PTP	Precise Time Protocol
RT	Real Time
RTC	Real Time
RTE	Real Time Ethernet
RTOS	Real Time Operating System
SM	Shared Medium
TCB	Task Control Block
TDMA	Time Division Multiple Access
TIOA	Timed Input Output Automata
TSC	Time Stamp Counter
TSIL	Time Slotted Interface Layer
URT	Urgency-based Real-time Protocol

CHAPTER 1

INTRODUCTION

Communication networks are commonly used at all organizational levels of industrial automation systems in the process control and manufacturing industry. They enable the coordination of the distributed supervisors with the exchange of local signals such as sensor or actuator data on either device or machine level.

Industrial communication networks provide a transmission medium for the signals of distributed supervisors. These supervisors have to send their messages within a specified deadline, consequently they are identified as real-time (RT) messages. These deadlines may vary depending on the control application needs; typically they are in the order of 10 ms and can be under 1 ms for closed-loop control applications. In addition, non real-time (nRT) messages may exist to serve as a diagnostic monitoring or maintenance. Those messages do not have a strict timing constraints.

The previous implementation of industrial communication protocols are all based on propriety fieldbuses. However, fieldbuses are not expandable and they are costly to operate as different corporations have fieldbuses incompatible with each other. Moreover, it is not practical to increase the network speed as the whole system has to be replaced with a new one. In contrast, Ethernet is a standard protocol that is highly available in home and office environments. Currently the available data rate for Ethernet is 100Gbps and it increases periodically.

Accordingly, the current trend is to employ Ethernet for industrial communication because of its high speed and low cost. However, the standard Ethernet (IEEE 802.3) cannot provide a real-time response for packet transmission because of the non-deterministic carrier sense multiple access with collision detection CSMA/CD arbitration algorithm. Therefore various Ethernet-based solutions are proposed with an additional RT support mechanism.

Studies in industry and academia proposed varying approaches for the development of Real-Time Ethernet (RTE). Those proposals are either making some modifications and additions to the network protocol stack of the conventional Ethernet protocol or employing switches with certain scheduling policies and traffic shaping. Most control applications requires proper synchronization of the controllers with some synchronization algorithm such as IEEE 1588 which also runs on Ethernet. A common drawback of the solutions in first category is that they provide real-time support on Ethernet by a static configuration of the possible sending instants or allocation of RT-bandwidth for each networked controller device. The problem with the switched implementation is the queueing delays in those switches, that undermines the synchronization accuracy. The delay may even grow up by cascaded switches. Next, those buffers in the switches have a limited capacity that may yield loss of packets in some cases.

In this thesis the implementation and experimental evaluation of Dynamic Distributed Dependable Real-Time Industrial communication Protocol (D^3RIP) [19], an Ethernet- based industrial communication protocol is presented. It consists of an interface layer that implements time division multiple access (TDMA) on top of the shared medium, and a coordination layer, that decides a unique controller for RT slots. D^3RIP dynamically allocates bandwidth to the RT messages with the information enclosed in the control application messages.

The implementation of D^3RIP presented in this thesis includes the coordination layer protocol family, time synchronization protocol and integration with the control application. In Figure 1.1 shaded parts illustrate the work done on the D^3RIP protocol stack. Before beginning programming the specified modules, we first configured each element of our system to run Realtime Linux with the best possible capabilities. The Linux kernel configuration was modified to allow real-time preemption and high-resolution timers were enabled. During the programming part, all the overheads that might be a possible source of latency were eliminated. In order to enable RT performance, paging was disabled, certain processes were given RT priorities and the use of global variables was avoided.

Coordination layer protocol implementation is carried out with an object-oriented software design. This enables further extension of this protocol without much effort in the future. In particular, two different versions of the coordination layer (DART and URT) were realized. The required priority queue within the coordination layer protocol is implemented by a binary heap structure together with a binary search algorithm which limited the computational cost of

queue operations to `O(log(N))`. Control application and the coordination layer were both residing on the user-space; thus implementing the interface was straightforward with a message queue. We used the control application `libFAUDES` [2] which is provided as an open-source project. We made a collaboration with the developer of `libFAUDES` and integrated the control application to send the real-time messages with required parameters for the coordination layer protocol. Two versions of the interface layer of D^{3}RIP (RAIL and TSIL) were developed by Korhan [1] within the kernel space. In cooperation with Korhan, the interface layer and the coordination layer were integrated. The difficulty of this step was that communication between the user-space and the kernel-space was required. Using a character device as an interfacing communication medium, the interprocess communication latency could be kept within the limits of the RT operating system. The implementation of IEEE 1588 time synchronization protocol and the clock servo that is used while accessing the system clock is based on the open-source project [3] and I further extended it utilizing the features of real-time operating system. There are no examples in the literature that are using a time-synchronization protocol running on a time-slotted Ethernet implementation. This potential delay impairs the accuracy of the resulting time synchronization. Time synchronization application sends non real-time packets to the interface layer, thus those packets may remain on the queue for some time if there are real-time packets to be sent. Consequently, we provide an additional interface between the IEEE 1588 application and the interface layer to make corrections on the acquired timestamps for outgoing packets.

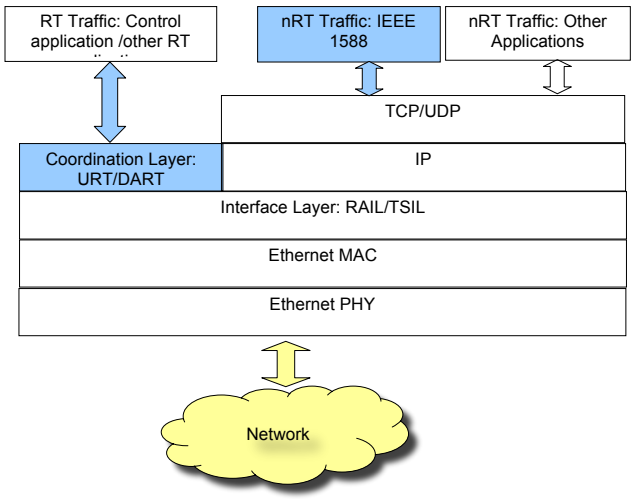

Figure 1.1: Implemented Part of the D^{3}RIP Protocol Stack

The remaining of the thesis is organized as follows. In Chapter 2 we describe the communication needs for control applications and review available industrial communication protocols. Chapter 3 reasons the design of the D^3RIP from control application communication needs. This chapter will be followed by a definition and specification of D^3RIP family. Before going into the implementation details, real-time operating systems and specifically Realtime Linux will be discussed in Chapter 5. Implementation details of IEEE 1588 time synchronization protocol is included in Chapter 6. Coordination layer protocol which is belonged to the D^3RIP family is implemented as given in Chapter 7. Then, we give brief implementation details for the integration of coordination layer protocol with the communication requests contained in the control application in Chapter 8. Chapter 9 contains the experiments and discussion about measurement results. Finally, Chapter 10 concludes the thesis.

CHAPTER 2

REAL-TIME INDUSTRIAL COMMUNICATION PROTOCOLS

Industrial control systems contain several components: sensors, Programmable Logical Controllers (PLC), Industrial PCs (IPC) and actuators [4]. These components coordinate through a networked communication. Increase in the demand of the industrial control applications resulted in larger and more complex control systems. . Consequently, industrial communication protocols drew more attention and it became an important industrial and academic research topic. Several industrial communication protocols were proposed within the last two decades [4],[5],[6],[7].

Industrial communication protocols distinguish messages for their purpose and they can be classified as follows: (Figure 2.1)

- T1: It involves the communication of the controllers with the equipments (sensors or actuators). Generally, the received data is periodic and requires to be delivered according to time constraints. Example: controller receives the speed data from a speed sensor, and sends the output current level to the actuator.
- T2: Supervisory control message provide the communication between system components that are at different hierarchical levels according to the hierarchical organization of the controllers and the controlled systems. These messages are usually triggered by occurrence of some events which makes them sporadic. Supervisory control messages require deterministic response times. For instance, a controller that controls two machines may send a message for the second machine when the first machine completes its current job.
- T3: Diagnostic messages provide the remote coordination of the control system. These

messages do not need strict response times and are often generated as a consequence of some events.

There are four requirements to realize the given communication messages:

- Real-Time traffic transfer: When a node prepares a real-time message (T1 and T2), it should be transferred before a specified deadline.
- Synchronized communication: All nodes should be synchronized with a common time-base to accomplish the transfer of real-time messages (T1 and T2).
- Dependability: There should be a dependability support for the possible faults or failures in the industrial control application.
- Support for non real-time traffic: These messages should be transferred over the network without a degradation in the real-time messages.

While satisfying the real-time guarantees in industrial communication networks, worst case assumptions should be considered. Similarly, dependability requirement to satisfy the given fault rates must be based on the worst case situations. For this reasons, the calculation of bandwidth allocation and bandwidth requirements are usually based on unrealistic assumptions such as transmission of all messages at the same time.

The initial communication protocols for industrial control systems are fieldbuses such as CAN, LonWorks and Profibus are started to be used twenty years ago [8]. These fielbuses are proprietary, costly, difficult to expand and are not compatible with other bus technologies. On the other hand, Ethernet (IEEE 802.3) is a simple, low cost and widely available option for the industrial communication protocols. Despite these advantages, generic Ethernet cannot provide real-time responses for the packets arriving at the same instant. This situation will result in a collision of packets and retransmission can be made on random time intervals. Consequently, Ethernet fails to provide deterministic transmission of the messages and cannot support dependable communication. In the recent years, several attempts are made as a topic of industrial and academic studies to develop Real-time Ethernet (RTE)[6], [7] .

Time Synchronization and Dependability:

Real-time control applications have requirements like: receiving a response at restricted times,

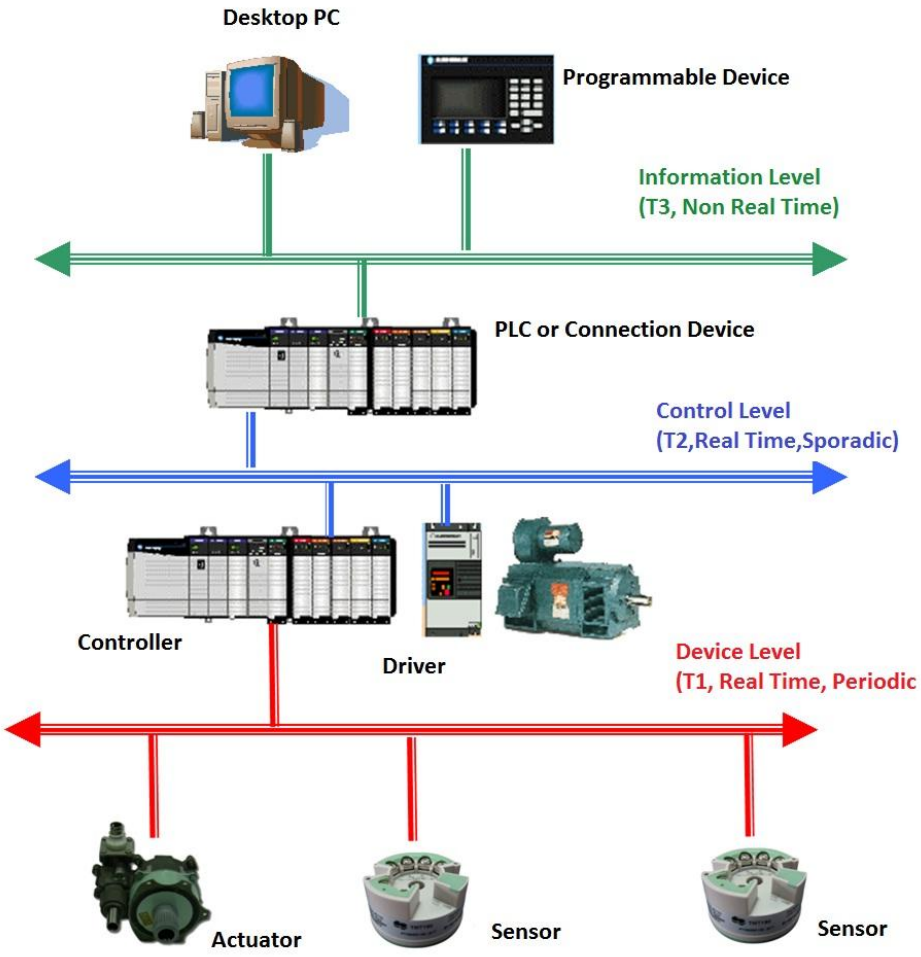

Figure 2.1: Communication Layers in Industrial Networks

minimum jitter for periodic events and following the correct sequence for the events. As these requirements exists on the whole distributed system, all of the nodes should be coordinated with a time synchronization [6], [7]. IEEE 1588 is a recent protocol that allows time synchronization for the nodes in a local area network. In this protocol a clock source is selected among the consisting nodes. Then the exchange of several messages provides time synchronization based on the selected clock source. There are two variants of the IEEE 1588 synchronization protocol: software only implementation may provide accuracy around 10-100us; whereas, including hardware modifications results in a synchronization performance in the order of nanoseconds. [6], [7], [9], [10].

Dependability is an important concern for the industrial control applications that have critical safety constraints [11]. It involves several attributes: availablility, safety, integrity and

maintainability [12]. Providing a dependable industrial communication system includes the dependability of controllers as well as the network. While designing a dependable industrial communication network, dependable distributed synchronization and the consistency of the information in the sent messages is crucial. Dependability problem is even more prominent for RTE based solutions due to the nondeterministic properties of Ethernet [13]. Dependable communication should ensure that the correct information is being sent to the right location at the right time and in the desired sequence. Dependability support is usually provided by considering the worst case situations [7]. For instance, TDMA based protocol allocates an extra bandwidth for each message that might be used in the case of a failure in the transmission. That yields only half of the capacity can be used at most.

Real Time Ethernet In Industrial Communication Protocols:
Ethernet has started to be developed in 1970s and the first standard for IEEE802.3 is published in 1985. In generic Ethernet, all nodes communicate over a shared medium; therefore, when one node transmits all others can detect the message. With this property Ethernet inherently provides the broadcast capabilities to the connected nodes. Ethernet node picks up the packet only if the destination address in the message specifies that node.

Collision occurs when two nodes sends a message to the shared medium at the same time. In order to prevent the situation, a node first listens to the medium and starts to transmit when there is no apparent transmission on the line. In the case where two nodes start to transmit at the same time, they will eventually stop transmission as they notice the collision. Before doing retransmission of those packets, nodes wait for a random amount of time. That random wait interval doubles up for each consequent collision. The random waiting interval before a retransmission violates the deterministic communication needs for industrial control applications.

There are essentially four practices in the literature to eliminate the non-deterministic behavior of Ethernet. Main approaches are; using specialized network interface cards altering the medium access behavior, minimizing collisions and increasing response times, using switches between peer-to-peer connections and constructing a layer on top of the shared medium to avoid collisions.

The operation of Ethercat [14], SERCOS III [15] and ProfiNet [16] protocols require specialized nodes and switches. Time synchronization in Ethercat and Profinet is accomplished by

IEEE 1588 and SERCOS III exchanges special messages for synchronization. Those three protocols comes with specially designed dependability protocols. In EtherCat, TwinSafe builds a protocol for dependability independent of the lower layers and the data security is handled by CRC. In SERCOS III Safety, messages contain a sequence number and timestamps. Receiving nodes send a ”message received” information to the sender node. Data security is provided by HDLC coding. Similarly, PROFIsafe enables message sequence numbers and timestamps for the Profinet protocol. Profinet uses CRC for data security.

MODBUS RTPS and PROFINET SRT runs at the application level to minimize the response time for real-time messages. However, there is no guarantee for the packets to arrive on time [7].

Switched Ethernet (IEEE 802.3x) is an option that avoids collision as it forms peer-to-peer full-duplex connection between nodes. Nodes are connected to each other over network switches and the multiple access to the shared medium is mapped to the queues within the switches [6], [7], [13]. Meeting the real-time operation requires switches with the capability of scheduling and prioritization [17]. Assigning priorities for the Ethernet packets and serving them differently are the subject of Ethernet protocols such as 802.1p, 802.1Q and their extensions. These protocols can only operate with specialized switches. Furthermore, the queues on the switches may overflow which results in packet loss.

Avoiding collisions is also possible by building a new layer on the shared medium and allowing a single node to transmit at a time. All packets either being real-time or non real-time pass through this layer. This layer may run under TCP-UDP/IP and possibly some other protocol as seen in Figure 2.2.

The additional protocol may regulate the access to the shared medium by Time Division Multiple Access (TDMA), Master-Slave architecture or Token passing.

TDMA: In this scheme the time slots are usually of equal length. Each node allocates several slots that can be used for message transmission. Thus TDMA requires a proper synchronization between nodes. The main disadvantage of TDMA is the inefficient bandwidth utilization. As the allocations are made statically, a node may reserve possibly unused time-slots. In addition, time slot duration should be chosen according to the variations of delays in the network and the synchronization accuracy. Additional time-slots should be reserved in the case of

packet losses. TTP/C protocol is an example for these TDMA solutions.

Master-Slave Architecture: Predefined node (master) polls other nodes (slaves) if they have a message to send. Slave nodes can only send messages if they are asked to. This technique is useful especially when the network traffic is periodic and the number of existing nodes is small. The throughput of the master-slave architecture depends on the time losses causes by polling of the slave nodes. For a network that involves excess amount of sporadic messages, the efficiency will decrease. Increasing the amount of nodes may cause an aggregate of polling periods to exceed the worst case latency of some messages. In that case, those messages will not be transmitted on the time. The computational latency of slave nodes is another factor that affects the speed of polling. As the response latency to polls increases the efficiency will decrease. In the end, execution time becomes a bottleneck instead of the network speed. Besides the efficiency outcomes, master-slave architecture is not suitable for distributed systems for single point of failure which is the master node. Sercos III and EtherCAT uses polling technique during operation.

Token Passing: A node can only send packets when it has the token that represent the right to transmit. The token circulates around the nodes as they send messages. Problems related with this technique are: losing tokens, lagging in the communication because of the token passing time and difficulty in adding new nodes to the system. TCnet protocol uses token-passing mechanism.

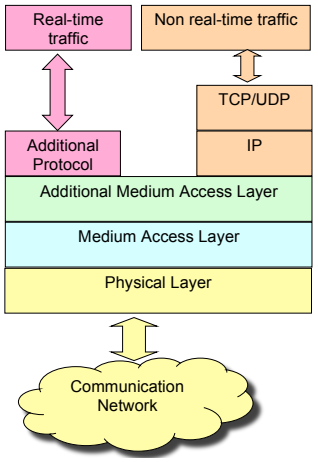

Figure 2.2: Real-Time Ethernet with Additional Medium Access Layer

CHAPTER 3

PROPERTIES OF THE CONTROL APPLICATIONS THAT MOTIVATE THE DESIGN OF D^{3}RIP

In this section, we discuss how to extract the runtime parameters needed by Dynamic Distributed Dependable Real-Time Industrial communication Protocol (D^{3}RIP) [19] from the control application. For simplicity, we give a control application example that runs with two supervisors which need to exchange some data for the proper system operation. We present the Communication Model for those supervisors that will reveal the communication requirements for the system.

3.1 Supervisor Synthesis in Discrete Event Systems

Discrete event system (DES) may describe many technical systems such as: automation plants, manufacturing systems and transport systems [20]. DES can be modeled by finite state automata models, and DES controllers, namely *supervisors*, are computed based on the automata model and a formal language specification. In most cases DES consists of several subsystems. Therefore, during the modeling process, these smaller components are treated separately and the overall system model required for the supervisor synthesis is computed by combining components. Those distributed controllers that are responsible for sub-systems synchronize with the events that are shared among different components

The decentralized controllers might be implemented with a physically distributed architecture. For example, an automobile manufacturing plant occupies multiple factory work floors and it imposes connected controllers to be far from each other. In this case the information about the occurrence of *shared events* has to be exchanged over a communication network, that

connects those distributed controllers.

3.2 Example System Consisting of Two Supervisors:

Figure 3.1 illustrates a simple discrete event system that is composed of four events. This example basically shows operation of a machine that may break and fail to operate at some time. Initially, the machine stays idle in `State 1`; when it receives event *op_start* it will start to operate by sending *op_inprog* event. During operation (`State 3`), machine can either break down or go back to idle state once again. As long as the device operates successfully, it can send the event *op_complete* which tells that the device operation completed successfully. Operation of the plant should be terminated by sending *op_stop* signal. Event *mu_request* however is sent by the plant telling that there is an operational failure in the machine during the operation. In that case, maintenance procedure is started externally by *mu_start* event and plant generates *mu_complete* signal when the machine is operable once again. Finally the maintenance procedure should be terminated by the sending *mu_stop* event to the plant.

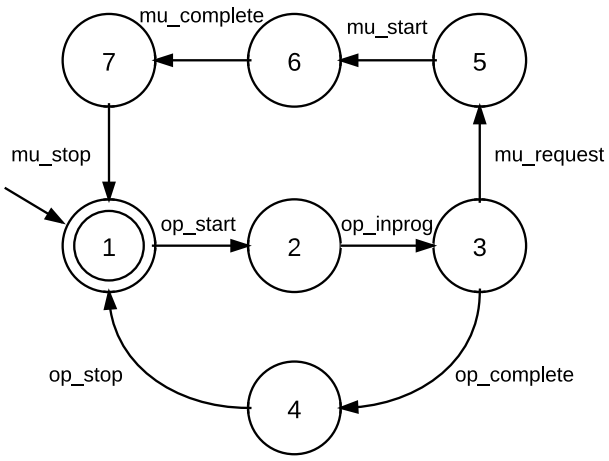

Figure 3.1: Operation of a Simple Machine

Simple machine given above example can be implemented by two distributed supervisors: G1 and G2. G1 operates according to Figure 3.2 such that, machine can be turned on and off by local events *op_start* and *op_stop* respectively. G1 should be connected to the machine with a serial interface so that it can send those actuator signals. In addition to that, G1 can receive an event μ from the other supervisor in the case of operational failure via D^3RIP.

G2 is basically responsible for sensing and repairing failures in the device operation as seen in Figure 3.3. It receives a sensor event *mu_request* when machine senses any malfunction during the operation. Then it initiates the maintenance process and informs the other supervisor with event μ that the machine is working once again. Note that there is no global naming of the states and the state names are local to each component.

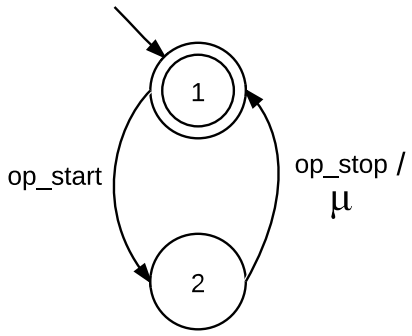

Figure 3.2: Logical Behavior of Supervisor G1

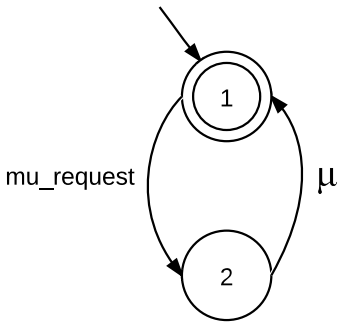

Figure 3.3: Logical Behavior of Supervisor G2

3.3 Communication Model:

Shared events are events that are assumed to happen at the same time in the model. In the example above the supervisors are physically separate so the occurence of the shared events must be synchronized by communication. In this thesis we make use of the automata-based communication model that is introduced in [18].

In principle, the communication model captures both the shared events between nodes and the

non-shared events that is devoted to some particular node. These events can be summarized as follows:

Shared events - Tasks: These are the events which organize the communication between system nodes and synchronize the operation of supervisors. In the simple machine example, G2 sends the shared event μ to G1 notifying that the machine is repaired.

Jobs: The synchronization of shared events is divided into several jobs that have to be exchanged among the distributed controller devices. To establish this communication, jobs are used as a container on the shared medium. For instance, if the repairing of the machine is the given task in the system, each message transmitted over the network to do the maintenance is defined as a job.

Non-Shared Events: They define local events that need not to be transported to other controllers. Non-shared events are generally sensor events that the controller receives from the plant or actuator events that plant accepts from the controller. For instance, *op_start* and *op_stop* events are the actuator signals that G1 can decide the machine to start or stop its operation respectively. These events should only be sent to the machine with a dedicated communication medium between machine and controller. Similarly, G2 realizes a malfunction in the machine by the sensor event *mu_request*.

The communication model ensures synchronization of shared events for the distributed controllers. The generated communication model CM1 for G1 is given in Figure 3.4 and for CM2 for G2 is illustrated in Figure 3.5. The jobs in both communication models are:

- $?\mu$: G2 sends this job to the G1 to check whether it is possible to execute the μ job in the G1.
- $!\mu$: After G1 receives $?\mu$ job from G2 when it is ready to execute μ event, it replies with $!\mu$ job telling that it can execute the specified job. However, G1 will not begin to execute the job before receiving μ job.
- μ: After receiving $!\mu$ job from G1, G2 knows that it is possible to execute the μ event. As a result it will send μ job to G1 to start the execution.

It is interesting to note that due to the deterministic definition of the supervisor operation it is possible to know the sequence of communicated jobs in advance. When G2 sends $?\mu$ to G1 it is known that G1 replies with $!\mu$ within some specified time.

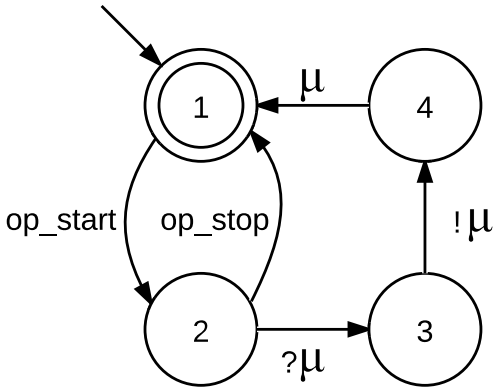

Figure 3.4: Communication Model for Controller G1

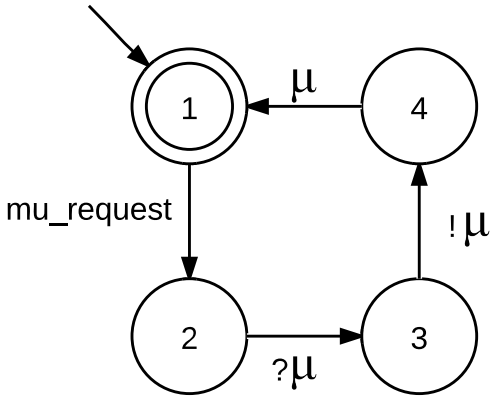

Figure 3.5: Communication Model for Controller G2

CHAPTER 4

D^{3}RIP OVERVIEW

Dynamic Distributed Dependable Real-Time Industrial communication Protocol (D^{3}RIP) is proposed as a new Real-Time Ethernet protocol with an aim of increasing bandwidth utilization [19]. This protocol is based on the fact that, the control application has the required information about communication needs of the distributed controller devices. D^{3}RIP uses this information to dynamically allocate the bandwidth according to the control application needs, and the rest of the resources can be available to all other non real-time applications. It is distinguished from other proposals in the literature (as stated in the previous chapter) that make statistical bandwidth allocations. D^{3}RIP consists of an Interface Layer (IL) and a Coordination Layer (CL) and can be implemented on conventional shared-medium Ethernet hardware as seen in Figure 4.1. Interface layer (IL) implements TDMA to allow the deterministic access of RT and nRT messages to the medium. Coordination layer (CL) lies on top of the IL and is responsible for the momentary allocation of RT slots to particular nodes on the network based on a distributed computation. Modeling of the IL and CL protocols is done using Timed I/O Automata (TIOA) to describe both discrete and continuous changes in time. There are generic IL and CL models which serve as basis for the construction of specific protocols.

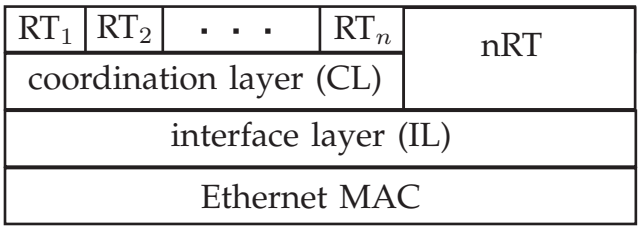

Figure 4.1: D^{3}RIP Protocol Architecture

4.1 Interface Layer

In this section we discuss the interface layer (IL) protocol family that provides time-slotted operation on top of a shared-medium broadcast channel such as Ethernet and below a coordination layer. It avoids possible collisions in the transmission of RT and nRT traffic by the unique assignment of time slots to network devices based on locally stored information and information that can be requested from the connected CL in each time slot.

4.1.1 Generic Interface Layer Model

General TIOA model of the IL (Figure 4.3) depicts the generic interface actions that are shared with the connected protocol layers in order to enable the data exchange. This model serves as a basis for two specific IL protocols that will be discussed next.

The TIOA model of the IL contains five characteristic parameters:

- Slot duration: $dSlot \in \mathbb{R}^+$
- Type of transmitted messages: M
- Type of a FIFO-queue of messages: Q
- Part of the time-slot that is not used for message transmission: *rem*
- Protocol related computations of IL: *cmp*

Figure 4.2 illustrates the relation between *rem* and *cmp*. Both of these parameters depends on the implementation of the protocol therefore they are left as variables in the work frame. A_{IL} abstract variable is used to indicate differences in two specialized protocols of the IL.

TIOA has variables for the local information that is needed during the operation. The analog variable $\texttt{now}_i^{\text{a}}$ evolves with a time derivative of 1 and provides the current time information to the nodes. Rest of the variables are all discrete. $\texttt{next}_i^{\text{d}}$ holds the end of the current time slot. The data structures used for the transmission and reception of the messages are $\texttt{TxRT}_i^{\text{d}}$, $\texttt{TxnRT}_i^{\text{d}}$, $\texttt{RxRT}_i^{\text{d}}$ and $\texttt{RxnRT}_i^{\text{d}}$ respectively. Structures used to store nRT messages are FIFO queues with type Q, while buffers for the RT messages can hold one element at most. For each time slot, $\texttt{RTIL}_i^{\text{d}}$ stores the next slot type that can either be a real-time or non-real time slot. This variable is complemented $\texttt{MyIL}_i^{\text{d}}$ that holds whether the device i owns the next time slot or not. Additional information for the protocol operation is stored in $\texttt{vIL}_i^{\text{d}}$. Internal functions f_{RT}, f_{my}, f_{req} and f_{upd} perform the updates of variables $\texttt{RTIL}_i^{\text{d}}$, $\texttt{myIL}_i^{\text{d}}$, $\texttt{reqIL}_i^{\text{d}}$ and $\texttt{vIL}_i^{\text{d}}$ respectively.

After the message transmission interval completes, `UPDATE` computes the protocol state for the consequent time slot. It computes $\texttt{vIL}_i^{\text{d}}$ and then updates $\texttt{reqIL}_i^{\text{d}}$. If IL does not need additional information for the next time slot ($\texttt{reqIL}_i^{\text{d}}$ =`false`), then the slot type becomes nRT slot ($\texttt{RTIL}_i^{\text{d}}$=`false`) and the ownership is determined locally. If there is a an information request for the next time slot ($\texttt{reqIL}_i^{\text{d}}$ =`true`), then it sends the REQRT$(t)_i$ message to the CL. CL2ILRT$(b_1, b_2, m)_i$ asks the type and the ownership of the next time slot providing the current time. CL uses the provided time information and sends the response CL2ILRT$(b_1, b_2, m)_i$ before the next slot. IL updates its state variables $\texttt{RTIL}_i^{\text{d}}$, $\texttt{myIL}_i^{\text{d}}$ with b_1 and b_2 respectively; it may also update $\texttt{TxnRT}_i^{\text{d}}$ if there is a RT message enclosed in the CL2ILRT$(b_1, b_2, m)_i$.

IL sends all messages destined to shared medium with IL2SM$(m)_i$ action. It can only happen in the beginning of next time slot and the ownership of the slot belongs to device i. Conversely, SM2IL(m) allows the message reception from the lower layer.

In each time slot, a time period $dSlot - rem$ is reserved for message transmission/reception. Rest of the slot duration is utilized for protocol related operations and computations. Assuming the message exchange happens in the defined intervals, RT messages are immediately forwarded to the upper layer. In addition, actions AP2ILNRT$(m)_i$ and IL2APNRT$(q)_i$ provides the exchange of nRT messages between IL and control application.

Collision-free operation on shared medium allows at most one interface layer to transmit in a given time slot. As IL proceeds with a distributed computation, it has to ensure that all protocol related parameters are the same after each update. $f_{RT}(\texttt{vIL}_i^d, b_1)$ function determines if the next time slot is a RT slot. CL provides b_1 variable and function decides consequent slot as a RT slot when b_1 is true.

$$f_{RT}(\texttt{vIL}_i^d, true) \quad = \quad \texttt{true} \tag{4.1}$$

IL decides owner of a consequent slot with $f_{my}(\texttt{vIL}_d^i, \texttt{RTIL}_d^i, b_2, i)$ where b_2 is provided by the upper layer. It should avoid the case in which several nodes owning a single time-slot.

$$\begin{aligned} f_{my}(\texttt{vIL}_d^i, \texttt{false}, -, i) \quad &= \quad \texttt{true} \\ \Rightarrow (\forall j \in I - \{i\}) \quad f_{my}(\texttt{vIL}_d^i, \texttt{false}, -, i) \quad &= \quad \texttt{false}, \end{aligned} \tag{4.2}$$

$$f_{my}(\texttt{vIL}_d^i, \texttt{true}, b_2, i) = b_2 \tag{4.3}$$

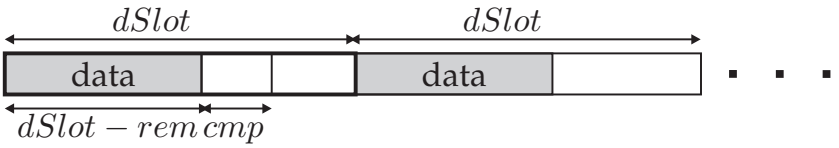

Figure 4.2: Typical Time Slot

TIOA $IL_i(dSlot, rem, cmp, M, Q, A_{\text{IL}})$

Variables X

$\texttt{now}_i^{\text{a}} \in \mathbb{R}$ (0)
$\texttt{next}_i^{\text{d}} \in \mathbb{R}$ ($dSlot$)
$\texttt{TxRT}_i^{\text{d}} \in M$ (empty)
$\texttt{TxnRT}_i^{\text{d}} \in Q$ (empty)
$\texttt{RxRT}_i^{\text{d}} \in M$ (empty)
$\texttt{RxnRT}_i^{\text{d}} \in Q$ (empty)
$\texttt{RTIL}_i^{\text{d}} \in \mathbb{B}$ (false)
$\texttt{myIL}_i^{\text{d}} \in \mathbb{B}$ (false)
$\texttt{vIL}_i^{\text{d}} \in A_{\text{IL}}$ (InitV)
$\texttt{reqIL}_i^{\text{d}} \in \mathbb{B}$ (false)

Actions A

input SM2IL$(m), m \in M$
input AP2ILNRT$(m)_i, m \in M$
input CL2ILRT$(b_1, b_2, m)_i$, $m \in M$, $b_1, b_2 \in \mathbb{B}$
input IL2APNRT$(q)_i, q \in Q$
output IL2CLRT$(m,t)_i$, $m \in M$, $t \in \mathbb{R}$
output IL2SM$(m)_i$, $m \in M$
internal UPDATE$_i$
output REQRT$(t)_i$, $t \in \mathbb{R}$

Transitions D

internal UPDATE$_i$
 precondition:
  $\texttt{now}_i^{\text{a}} = \texttt{next}_i^{\text{d}} - rem$
  $\texttt{RxRT}_i^{\text{d}}$ empty
 effect:
  $\texttt{vIL}_i^{\text{d}} = f_{\text{upd}}(\texttt{vIL}_i^{\text{d}}, \texttt{RTIL}_i^{\text{d}})$
  $\texttt{reqIL}_i^{\text{d}} = f_{\text{req}}(\texttt{vIL}_i^{\text{d}})$
  if $\neg\texttt{reqIL}_i^{\text{d}}$
   $\texttt{RTIL}_i^{\text{d}} =$ false
   $\texttt{myIL}_i^{\text{d}} = f_{\text{my}}(\texttt{vIL}_i^{\text{d}}, \texttt{RTIL}_i^{\text{d}}, b_2, i)$
  $\texttt{next}_i^{\text{d}} = \texttt{next}_i^{\text{d}} + dSlot$

output IL2CLRT$(m, \texttt{now}_i^{\text{a}})_i$
 precondition:
  $\texttt{now}_i^{\text{a}} = \texttt{next}_i^{\text{d}} - rem$
  $\neg(\texttt{RxRT}_i^{\text{d}}$ empty$)$
 effect:
  set $m = \texttt{RxRT}_i^{\text{d}}$
  set $\texttt{RxRT}_i^{\text{d}}$ empty

output REQRT$(\texttt{now}_i^{\text{a}})_i$
 precondition:
  $\texttt{reqIL}_i^{\text{d}} =$ true
  $\texttt{now}_i^{\text{a}} = \texttt{next}_i^{\text{d}} - dSlot - rem + cmp$
 effect:
  $\texttt{reqIL}_i^{\text{d}} =$ false

input IL2APNRT$(\texttt{RxnRT}_i^{\text{d}})_i$
 effect:
  set $\texttt{RxnRT}_i$ empty

input SM2IL(m)
 effect:
  if $\texttt{RTIL}_i^{\text{d}}$
   $\texttt{RxRT}_i^{\text{d}} = m$
  else
   $\texttt{RxnRT}_i^{\text{d}}$.Push$(m)$

output IL2SM$(m)_i$
 precondition:
  $(\texttt{now}_i^{\text{a}} = \texttt{next}_i^{\text{d}} - dSlot) \wedge \texttt{myIL}_i^{\text{d}}$
  $(\neg(\texttt{TxRT}_i^{\text{d}}$ empty$) \wedge \texttt{RTIL}_i^{\text{d}}) \vee (\neg\texttt{RTIL}_i^{\text{d}} \wedge \neg(\texttt{TxnRT}_i^{\text{d}}$.Top empty$))$
 effect:
  if $\texttt{RTIL}_i^{\text{d}}$
   set $m = \texttt{TxRT}_i^{\text{d}}$
   set $\texttt{TxRT}_i^{\text{d}}$ empty
  if $\neg\texttt{RTIL}_i^{\text{d}}$
   set $m = \texttt{TxnRT}_i^{\text{d}}$.Top
   $\texttt{TxnRT}_i^{\text{d}}$.Pop
  $\texttt{myIL}_i^{\text{d}} =$ **false**

input CL2ILRT$(b_1, b_2, m)_i$
 effect:
  $\texttt{RTIL}_i^{\text{d}} = f_{\text{RT}}(\texttt{vIL}_i^{\text{d}}, b_1)$
  $\texttt{myIL}_i^{\text{d}} = f_{\text{my}}(\texttt{vIL}_i^{\text{d}}, \texttt{RTIL}_i^{\text{d}}, b_2, i)$
  $\texttt{TxRT}_i^{\text{d}} = m$

input AP2ILNRT$(m)_i$
 effect:
  $\texttt{TxnRT}_i^{\text{d}}$.Push$(m)$

Trajectories $\mathcal{T}$

stop when
 $\texttt{now}_i^{\text{a}} = \texttt{next}_i^{\text{d}} - dSlot \wedge \texttt{myIL}_i^{\text{d}}$
 $\texttt{now}_i^{\text{a}} = \texttt{next}_i^{\text{d}} - rem$
 $\left(\texttt{now}_i^{\text{a}} = \texttt{next}_i^{\text{d}} - dSlot - rem + cmp\right) \wedge \texttt{reqIL}_i^{\text{d}}$

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

Figure 4.3: IL Model as a TIOA

For the correct operation of IL, upper layers should satisfy several conditions. These external requirements are as follows:

Axiom 1 (External Requirements for interface layer)

- All messages fit into the transmission window.

$$m.length < dSlot - rem \quad \forall m \in M \tag{4.4}$$

- CL2ILRT$_i$ occurs exactly once after each request REQRT$_i$. The time between REQRT$_i$ and CL2ILRT$_i$ is bounded by $rem - cmp$.

- If REQRT$(t)i$ and REQRT$(t)_j$ $i, j \in I$, $i \neq j$ occur at the same time t, then, it holds for the next occurrence of

  CL2ILRT$(b_1, b_2, m)_i$ and CL2ILRT$(\hat{b_1}, \hat{b_2}, \hat{m})_j$ that $b_1 = \hat{b_1}$ and $b_2 = true \Rightarrow \hat{b_2} = false$

4.1.2 Real-time Access Interface Layer (RAIL)

RAIL makes statistical slot allocations for RT and nRT messages while accessing shared medium. The variable *cyc* is defined for periodic time slot allocation. $RTSet \subseteq 0, 1, ..., cyc - 1$ set is for real-time messages and `nRTSet` is for the time slots that can be used for nRT messages by the interface layer IL$_i$. The relationship between these sets is defined as; $\texttt{vIL}_i^{\text{d}}.\texttt{RTSet} \cap \texttt{vIL}_i^{\text{d}}.\texttt{nRTSet} = \emptyset$ for $i \in \mathcal{I}$ and $\texttt{vIL}_i^{\text{d}}.\texttt{nRTSet} \cap \texttt{vIL}_j^{\text{d}}.\texttt{nRTSet}_j = \emptyset$ for $i, j \in \mathcal{I}$, $i \neq j$.

The data structure $\texttt{vIL}_i^{\text{d}}$ used in implementation is composed of slot counter $\texttt{vIL}_i.\texttt{cnt}$ and statical variables, $\texttt{vIL}_i.\texttt{cyc}$, $\texttt{vIL}_i^{\text{d}}.\texttt{RTSet}$ and $\texttt{vIL}_i^{\text{d}}.\texttt{nRTSet}$. Those parameters are initialized by $\texttt{vIL}_i.\texttt{cyc} = 0$, $\texttt{vIL}_i^{\text{d}}.\texttt{RTSet}$= RT Set and $\texttt{vIL}_i^{\text{d}}.\texttt{nRTSet} = nRTSet$. After the initialization, f_{upd} increments slot counter modulo $\texttt{vIL}_i.\texttt{cyc}$ in each time slot such that the slot assignment repeats every *cyc* slots and the others remain unchanged. Allocation of time-slots repeat after each *cyc*. When the value of *cnt* is an element of RT Set, IL sends request, f_{upd} to the CL. If the CL decides the next slot to be RT slot ($b_1 = true$), CL2ILRT$(true, -, -)_i$ will be received and the internal function f_{RT} returns true (4.1). For RT slots, owner of the slot is determined by b_2 variable received from CL, where, IL determines the owner of nRT slots by looking the nRT Set and the slot counter value *cnt*.

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

The implemented functions satisfy all the requirements given in (4.1), (4.2) and (4.3). Thus, RAIL exhibits all properties given in the generic interface layer as long as the upper protocol layer fulfills the external requirements.

When a node has a RT message to send, it has to wait for its next available RT slot. We will compute the number of time slots $\#_{RAIL}(n)$ that the device has to wait, assuming that it owns the n-th nearest RT slot. For this purpose, we introduce $w_{RAIL}(m,n)$: the known number of time slots that pass until the n-th RT slot is reached starting with the time slot counter value m. Number of time slot that a node has to wait is:

$$
\#_{\text{RAIL}}(n) = \max_{0 \le m < \texttt{vIL}_i.\texttt{cyc}} w_{\text{RAIL}}(m,n). \tag{4.5}
$$

One of the most important characteristics of the RAIL is it that, upper layer protocols may use RT slots freely, while the separation of RT and nRT slots is made statically. The statical separation of time-slots eliminates possible sources of error while making time-slot type decision. Major drawback of such static separation is the reduced efficiency of the bandwidth use. RAIL assigns a RT bandwidth to control application any time has to be assigned however, it might not be necessary at all times.

4.1.3 Time-slotted Interface Layer (TSIL)

Time-slotted Interface Layer (TSIL) enables coordination layer to decide the next slot-type as RT or nRT dynamically. We define the decision variables $\texttt{vIL}_i^{\text{d}}.\texttt{cnt}$, $\texttt{vIL}_i^{\text{d}}.\texttt{cyc}$ and $\texttt{vIL}_i^{\text{d}}.\texttt{nRTSet}$ similar to RAIL. However, slot count only increments on nRT slots and f_{req} always returns

true such that the upper layer can always determine next-slot type using f_{RT}. f_{my} is again uses the variable b_2 of upper layer to decide owner of RT slots, while the ownership of nRT slots is locally decided by TSIL.

$$f_{\mathrm{upd}}(\mathtt{vIL}_i^{\mathrm{d}}, \mathtt{RTIL}_i^{\mathrm{d}}).\mathtt{cnt} = \begin{cases} \mathtt{vIL}_i^{\mathrm{d}}.\mathtt{cnt} & \text{if } \mathtt{RTIL}_i^{\mathrm{d}} \\ (\mathtt{vIL}_i^{\mathrm{d}}.\mathtt{cnt} + 1) & \\ \mathrm{mod}\ \mathtt{vIL}_i^{\mathrm{d}}.\mathtt{cyc} & \text{otherwise} \end{cases}$$

$$f_{\mathrm{req}}(\mathtt{vIL}_i^{\mathrm{d}}) = \mathrm{true}$$

$$f_{\mathrm{RT}}(\mathtt{vIL}_i^{\mathrm{d}}, b_1) = \begin{cases} \mathrm{true} & \text{if } b_1 = \mathrm{true} \\ \mathrm{false} & \text{otherwise} \end{cases}$$

$$f_{\mathrm{my}}(\mathtt{vIL}_i^{\mathrm{d}}, \mathtt{RTIL}_i^{\mathrm{d}}, b_2, i) = \begin{cases} b_2 & \text{if } \mathtt{RTIL}_i^{\mathrm{d}} = \mathrm{true} \\ \mathrm{true} & \text{if } \neg\mathtt{RTIL}_i^{\mathrm{d}} \wedge \mathtt{vIL}_i^{\mathrm{d}}.\mathtt{cnt} \\ & \in \mathtt{vIL}_i^{\mathrm{d}}.\mathtt{nRTSet} \\ \mathrm{false} & \text{otherwise.} \end{cases}$$

By the implementation details given above, the equations 4.1, 4.2 and 4.3 are all fulfilled. As the upper layer protocol dynamically decides on the slot types, TSIL can better adapt to the instantaneous RT communication requirements of control applications. Actually, a device that wants to use the n-th closest slot as an RT slot will be able to send message in the n-th nearest slot.

$$\#_{\mathrm{TSIL}}(n) = n. \tag{4.6}$$

Operation of TSIL does not require any additional configuration. Slots that are not allocated for the RT messages can be used by nRT messages during the operation, hence RAIL implementation increases the efficiency in bandwidth utilization especially for nRT messages.

4.2 Coordination Layer

This section will describe the coordination layer (CL) protocol family that can work on top of the IL. In this manner, CL has to fulfill the external requirements of IL in Axiom 1 and relies on the time-slotted medium access provided by IL. During its operation, the CL of each device processes and broadcasts information from the control application to the other devices using RT messages. Accordingly, the CLs of the different devices can adjust their RT behavior during operation based on a distributed computation.

4.2.1 Generic Coordination Layer Model

The coordination layer TIOA CL_i for each device i is shown in Figure 4.4. Its parameters are:

- Processing *delay* value: del_i
- Message type M that offers M.data for message data and M.par for protocol related parameters
- FIFO-queue of messages with type Q and the type V that represents a vector of messages with type M
- Abstract data type: A_{CL} varies for different members of the CL protocol family

The CL stores all messages in either of the discrete variables $\mathtt{Tx}_i^{\mathrm{d}}$ and $\mathtt{Rx}_i^{\mathrm{d}}$. It has to support the transmission of messages for different *channels*. $\mathtt{Tx}_i^{\mathrm{d}}$ is realized as a vector of message buffers, where each entry can hold the current message for one *channel*. The ownership of RT slots is stored in the boolean variables $\mathtt{RTCL}_i^{\mathrm{d}}$ and $\mathtt{myCL}_i^{\mathrm{d}}$ and in the channel variable $\mathtt{ch}_i^{\mathrm{d}}$. Update of these variables depends on decision variables $\mathtt{vCL}_i^{\mathrm{d}} \in A_{\mathrm{CL}}$. The only analog variable is $\mathtt{send}_i^{\mathrm{a}}$ and it is used to store the elapsed time after a request was issued from the IL.

CL builds a decision mechanism for subsequent time-slots upon receiving request from IL. While doing so, CL also computes a unique sender for RT slots to avoid collision. During computation, CL utilizes the decision variables $\mathtt{vCL}_i^{\mathrm{d}}$, $\mathtt{reqCL}_i^{\mathrm{d}}$, $\mathtt{RTCL}_i^{\mathrm{d}}$, $\mathtt{myCL}_i^{\mathrm{d}}$, $\mathtt{ch}_i^{\mathrm{d}}$ as well as timing information t and the RT message parameters $m.\mathtt{par}$ that are provided by the IL.

When the CL receives a request from IL ($\textsc{reqrt}(t)_i$) it updates the status $\mathtt{RTCL}_i^{\mathrm{d}}$ and the ownership ($\mathtt{myCL}_i^{\mathrm{d}}, \mathtt{ch}_i^{\mathrm{d}}$) of the subsequent time slot. If the computation time $\mathtt{send}_i^{\mathrm{a}}$ does not exceed del_i, this slot information is transmitted to IL with the action $\textsc{cl2ilrt}_i(\mathtt{RTCL}_i^{\mathrm{d}}, \mathtt{myCL}_i^{\mathrm{d}}, m)$. Each device i can only provide non-empty message data if it is the unique sender device with $\mathtt{myCL}_i^{\mathrm{d}} = \text{true}$. The operation of CL is independent from the evolution of time. The variable $\mathtt{send}_i^{\mathrm{a}}$ is only used to formally model that the action $\textsc{cl2ilrt}_i$ occurs at most del_i time units after $\textsc{reqrt}_i$, where it is assumed that $del_i < rem - cmp$

CL receives RT messages from upper layer by $\textsc{ap2cl}_i$ and from lower layer by $\textsc{il2clrt}_i$ actions. In the first case, the message in $\mathtt{Tx}_i^{\mathrm{d}}[ch]$ is prepared for transmission from the data *dat* and the protocol parameters *par*. In the latter case, the message is stored in $\mathtt{Rx}_i^{\mathrm{d}}$ and the decision variables are updated based on the protocol parameter information $m.\mathtt{par}$ and the timing information t received from the IL. The control application can reach message data with the action $\textsc{cl2ap}_i$.

The definition of CL in Figure 4.4 is based on the functions g_{upd}, g_{RT} and g_{my} that determine the current values of the decision variables $\mathtt{vCL}_i^{\mathrm{d}}$, the slot type $\mathtt{RTCL}_i^{\mathrm{d}}$ and the slot ownership with the corresponding channel $(\mathtt{myCL}_i^{\mathrm{d}}, \mathtt{ch}_i^{\mathrm{d}})$, respectively. The requirement is that CL_i can only use RT slots whose ownership is uniquely determined among the coordination layers CL_i, $i \in \mathcal{I}$.

$$g_{\text{my}}(\mathtt{vCL}_i^{\mathrm{d}}, \text{false}, t, i) = (\text{false}, 0),$$

$$g_{my}(\mathtt{vCL}_i^{\mathrm{d}}, \text{true}, t, i) = (\text{true}, ch)$$

$$\Rightarrow g_{my}(\mathtt{vCL}_i^{\mathrm{d}}, \text{true}, t, j) = (\text{false}, 0) \text{ for all } j \in \mathcal{I} - \{i\}$$

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

Variables X

$\mathtt{send}_i^{\mathrm{a}} \in \mathbb{R}$ (del_i)
$\mathtt{Tx}_i^{\mathrm{d}} \in V$ (empty)
$\mathtt{Rx}_i^{\mathrm{d}} \in Q$ (empty)
$\mathtt{RTCL}_i^{\mathrm{d}} \in \mathbb{B}$ (false)
$\mathtt{myCL}_i^{\mathrm{d}} \in \mathbb{B}$ (false)
$\mathtt{ch}_i^{\mathrm{d}} \in \mathbb{N}$ (0)
$\mathtt{reqCL}_i^{\mathrm{d}} \in \mathbb{B}$ (false)
$\mathtt{vCL}_i^{\mathrm{d}} \in A_{\text{CL}}$ $(InitCL)$

Actions A

input AP2CL$(dat, p, ch)_i$, $dat \in M$.data, $p \in M$.par, $ch \in \mathbb{N}$
input CL2AP$(q)_i$, $q \in Q$
input IL2CLRT$(m,t)_i$, $m \in M$, $t \in \mathbb{R}$
input REQRT$(t)_i$, $t \in \mathbb{R}$
output CL2ILRT$(\mathtt{RTCL}_i^{\mathrm{d}}, \mathtt{myCL}_i^{\mathrm{d}}, m)_i$

Transitions D

input AP2CL$(dat, p, ch)_i$
effect:
$\mathtt{Tx}_i^{\mathrm{d}}[ch].\mathtt{data} = dat$
$\mathtt{Tx}_i^{\mathrm{d}}[ch].\mathtt{par} = p$

input REQRT$(t)_i$
effect:
$\mathtt{RTCL}_i^{\mathrm{d}} = g_{RT}(\mathtt{vCL}_i^{\mathrm{d}}, \mathtt{RTCL}_i^{\mathrm{d}}, t)$
$(\mathtt{myCL}_i^{\mathrm{d}}, \mathtt{ch}_i^{\mathrm{d}}) = g_{my}(\mathtt{vCL}_i^{\mathrm{d}}, \mathtt{RTCL}_i^{\mathrm{d}}, t, i)$
$\mathtt{send}_i^{\mathrm{a}} = 0$
$\mathtt{reqCL}_i^{\mathrm{d}} = \text{true}$

input CL2AP$(\mathtt{Rx}_i^{\mathrm{d}})_i$
effect:
set $\mathtt{Rx}_i^{\mathrm{d}}$ empty

input IL2CLRT$(m,t)_i$
effect:
$\mathtt{Rx}_i^{\mathrm{d}}$.Push$(m)$
$\mathtt{vCL}_i^{\mathrm{d}} = g_{\text{upd}}(\mathtt{vCL}_i^{\mathrm{d}}, m.\mathtt{par}, t)$

output CL2ILRT$(\mathtt{RTCL}_i^{\mathrm{d}}, \mathtt{myCL}_i^{\mathrm{d}}, m)_i$
precondition:
$\mathtt{reqCL}_i^{\mathrm{d}} \wedge (\mathtt{send}_i^{\mathrm{a}} \leq del_i)$
effect:
if $\mathtt{myCL}_i^{\mathrm{d}}$
  set $m = \mathtt{Tx}_i^{\mathrm{d}}[\mathtt{ch}_i^{\mathrm{d}}]$
  set $\mathtt{Tx}_i^{\mathrm{d}}[\mathtt{ch}_i^{\mathrm{d}}]$ empty
else
  set m empty
$\mathtt{reqCL}_i^{\mathrm{d}} = \text{false}$

Trajectories $\mathcal{T}$

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

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

Figure 4.4: CL Model as a TIOA

4.2.2 Dynamic Allocation Real-time protocol (DART)

In dynamic allocation real-time (DART) protocol the variables are stored in the form of allocated RT slots that are assigned to specific controller devices. This information is dynamically updated by the parameters received from control application. The decision variables contain

a cycle variable $\mathrm{vCL}_i^{\mathrm{d}}$.cyc and an RT slot counter $\mathrm{vCL}_i^{\mathrm{d}}$.cnt. In addition, the *allocation data object* (ado) is introduced with the parameters ado.num $\in \mathbb{N}$, ado.slots $\subseteq \{0, \ldots, cyc - 1\}$, ado.used $\in \mathbb{N} \times \mathbb{N}$. ado.num describes the number of RT slots allocated per *cyc* RT slots, ado.slots holds an ordered list of ado.num allocated RT slots and ado.used is a tuple that indicates the device that currently uses the allocation data object and its channel Id ($(0, 0)$ if it has no usage). Each CL_i contains a vector $\mathrm{vCL}_i^{\mathrm{d}}$.alloc of allocation data objects. The usage of those slots should be mutually exclusive at different nodes. Thus, It has to hold that alloc[k].slots $\neq$ alloc[l].slots for $k \neq l$.

In DART, the parameters of the message received from control application should consist two fields: par.free and par.new. The first one is used to free the allocated time-slot resources that are found in the *alloc*[k].*used* list. Control application indicates the requirement to allocate new channels by triple $(a, b, c) \in \mathbb{N} \times \mathbb{N} \times \mathbb{N}$. In here, the first element a tells the required amount of slots for the node b at channel c. Update functions for DART are:

$$g_{\mathrm{upd}}(\mathrm{vCL}_i^{\mathrm{d}}, \mathrm{m.par}, t)) = (\mathrm{vCL}_i^{\mathrm{d}}.\mathrm{cnt} + 1) \mod \mathrm{vCL}_i^{\mathrm{d}}.\mathrm{cyc}$$

$$g_{\mathrm{RT}}(\mathrm{vCL}_i^{\mathrm{d}}, \mathrm{RTCL}_i^{\mathrm{d}}, t) = \begin{cases} \text{true} & \text{if } \mathrm{vCL}_i^{\mathrm{d}}.\mathrm{cnt} \in \mathrm{vCL}_i^{\mathrm{d}}.\mathrm{alloc}[k].\mathrm{slots} \\ & \text{for some } k \\ \text{false} & \text{otherwise} \end{cases}$$

$$g_{\mathrm{my}}(\mathrm{vCL}_i^{\mathrm{d}}, \mathrm{RTCL}_i, t, i) = \begin{cases} (\text{true}, c) & \text{if } \mathrm{vCL}_i^{\mathrm{d}}.\mathrm{cnt} \in \mathrm{vCL}_i^{\mathrm{d}}.\mathrm{alloc}[k].\mathrm{slots} \\ & \wedge \mathrm{vCL}_i^{\mathrm{d}}.\mathrm{alloc}[k].\mathrm{used} = (i, c) \\ (\text{false}, 0) & \text{otherwise} \end{cases}$$

In addition, $g_{\mathrm{upd}}(\mathrm{vCL}_i^{\mathrm{d}}, m.\mathrm{par}, t)$ updates the allocation data objects such that each entry in m.par.free is removed from the list ado.used whereas we add new ones to the same list according to the m.par.new.

4.2.3 Urgency-Based Real-time Protocol (URT)

Urgency-based real-time protocol (URT) stores the control application parameters in the form of *communication requests*. The only element of the decision variable will be priority queue $\mathrm{vCL}_i^{\mathrm{d}}$.PQ and it holds the *communication requests* in the form of a 4-tuple (b, c, eT, dT) in which b denotes a device, c is a channel, eT is an *eligibility time* and dT is a *deadline* measured relative to the time instant where the request is issued. The *communication request* indicates

that, device b can send the next message of the channel c after time eT and must send the next message before dT. Consistently, control application message contains set of requests m.par.req as its protocol parameter. After receiving a control application message, the requests are pushed into $\mathtt{vCL}_i^{\mathrm{d}}$.PQ, $i \in \mathcal{I}$, ordered by eligibility time and deadline such that in each time slot, the device with the most urgent eligible request gets access to the medium. It is seen that urgency of the requests are determined by control application.

Update functions for URT are defined as follows; if $g_{\mathrm{upd}}(\mathtt{vCL}_i^{\mathrm{d}}, m.\mathrm{par}, t)$ is called and $\mathtt{RTCL}_i^{\mathrm{d}} =$ true, the first request is popped from $\mathtt{vCL}_i^{\mathrm{d}}$.PQ if a RT message is received. If no RT message appears, the first request reenters $\mathtt{vCL}_i^{\mathrm{d}}$.PQ to provide transmission in the following time-slots. In addition, requests in *par*.req are inserted in $\mathtt{vCL}_i^{\mathrm{d}}$.PQ after the current time t is added to the relative times in each request. The remaining function definitions are:

$$g_{\mathrm{RT}}(\mathtt{vCL}_i, \mathtt{RT}_i, t) = \begin{cases} \text{true} & \text{if } \mathtt{vCL}_i.\mathtt{PQ}.\mathrm{Top}.eT \leq t \\ \text{false} & \text{otherwise} \end{cases}$$

$$g_{\mathrm{my}}(\mathtt{vCL}_i, \mathtt{RT}_i, t, i) = \begin{cases} (\text{true}, a) & \text{if } \mathtt{PQ}_i.\mathrm{Top}.b = i \wedge \mathtt{RT}_i \\ & = \text{true} \wedge \mathtt{PQ}_i.\mathrm{Top}.c = a \\ (\text{false}, 0) & \text{otherwise} \end{cases}$$

URT can be used with both TSIL and RAIL. For TSIL, it holds that every time slot can be used as an RT slot if required. It is shown in [18] that every request meets its deadline if,

$$dS\,lot \leq \frac{dT_{\min} - eT_{\max}}{PQ_{\max} + 1}, \tag{4.7}$$

where $dT_{\min}$ is the minimum request deadline, $eT_{\max}$ is the maximum eligibility time and $PQ_{\max}$ is the maximum length of the priority queue. For RAIL, it has to be considered that each slot is not available for RT traffic. Using the result in (4.5), the above equation will be,

$$dS\,lot \leq \frac{dT_{\min} - eT_{\max}}{(PQ_{\max} + 1) \cdot \#_{\mathrm{RAIL}}(PQ_{\max})}, \tag{4.8}$$

CHAPTER 5

REAL-TIME OPERATING SYSTEMS and REALTIME LINUX

Implementing the D^3RIP protocol on the target hosts requires real-time guarantees. While programming real-time software, one has to know the abilities and capabilities of Realtime Operating System that is being used. This section begins with distinctions between Real-Time Operating System (RTOS) and daily-use operating systems. Next, we discuss the proper metrics for measuring RTOS performance and Realtime Linux is covered afterwards.

5.1 Real-Time Operating System Fundamentals

As the daily-use operating systems are developed for providing the best performance for the general case, they fail to offer deadlines for real-time software. This is why RTOSs are designed: to serve real-time applications on time. RTOS serve applications within some guaranteed time bounds. They build a framework for the real-time processes to meet timing constraints. A RTOS must provide deterministic responses to unpredictable concurrent events. An application that runs on a RTOS does not necessarily obey the real-time constraints. Meeting the deadlines and response times needs a careful design. The RTOS and the processes should run in a harmony. The designer should know the limits of the RTOS and also be aware of the running real-time processes during the system operation. Embedded processors are generally the target platform to run RTOS, however, it is possible to run some RTOSs on desktop processors such as Intel x86 family. RTOS share some common properties with the standard operating systems, however several characteristics make the distinction. Essentially, they provide the desired abstraction between the hardware and upper level application layer. RTOSs offer the following services similar to the general-usage operating systems:

- File Systems

- Networking
- Device Drivers
- Security

What makes an RTOS special is the difference in the essentials of the operating system that may cause unbounded latencies. RTOS has significantly different approach in the following services:

- Process Management
- Scheduling
- Context Switching
- Interprocess Communication
- Interrupt Handling
- Memory Management

Before going into the details of the RTOS, it is intuitive to cover RTOS classification. RTOSs are divided into two sub-categories according to their reaction to real-time constraints. *Soft-RTOSs* are designed to meet the deadlines most of the time and the processing performance is generally an important concern for this type. Percentage of missed deadlines is a possible metric for benchmarking. *Hard-RTOSs* specify deterministic response times and meet them on the long run. Context switching time and interrupt latencies are among the performance metrics for Hard-RTOS. If missing any deadlines causes a system to fail, then hard-real time OS will definitely be the choice.

Process Management: Central processing unit (CPU) is able to perform one instruction of any process at a time. In modern operating systems, such as Linux and Microsoft Windows, several processes may reside in the memory and wait for the CPU to perform their instructions. CPU serves those processes at some intervals determined by the operating system. In this way, multiple processes may be executed at a particular time instant. Operating systems that are capable of concurrently executing several processes are known as *multi-tasking operating systems*. It increases the processor utilization over the single-tasking operating systems in which CPU may lay idle considerable amount of time. Figure 5.1 illustrates the process states and transitions in a multitasking RTOS [21].

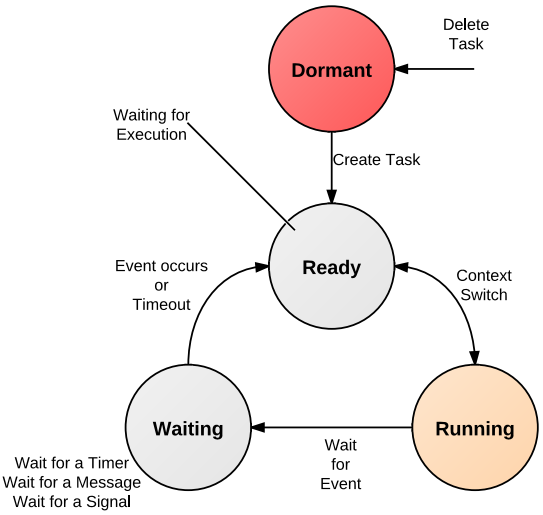

Figure 5.1: Process States in RTOS

When a user creates and starts a process, it will initially be in the Ready state. Each process will have corresponding Task Control Block (TCB) indicating the related information about the particular process (process identifier, register values, etc.). Process in the Ready state resides on a linked list so-called Ready-List as seen in Figure 5.2 which stores multiple processes in order that are waiting to execute their instructions [21]. A process is at Running state when CPU executes its instruction. Waiting process stays blocked until it receives a timer interrupt, a message or a signal. After any of the interrupts wakes the process up, it immediately enters into the Ready List. Process in the Dormant state resides in memory inactively.

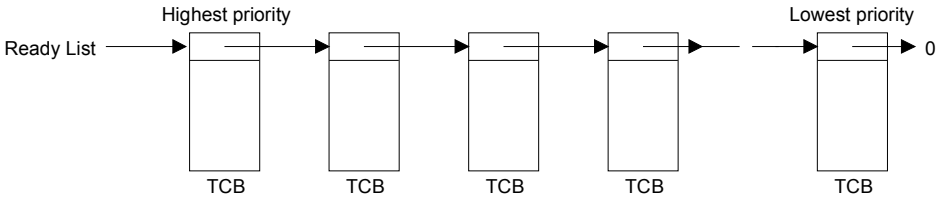

Figure 5.2: Ready List

Scheduling: In general-purpose operating systems, processes are executed fairly. That means a process in the Running state grasps the CPU for a fixed time-slice. It may end up its execution before time-slice when it finishes up its instructions or needs another system resource to continue. The scheduler is generally implemented with a round-robin algorithm with time-slice. This type of scheduling may cause the real-time processes to miss their deadlines for the sake of fairness. RTOS resolves this problem by introducing the *Preemption Policy* to

the scheduler. In this case, scheduler may preempt a lower-priority task in the Running state and put a higher priority one instead. RTOS continuously polls the Ready List to check if there is a higher priority task waiting in the queue. It is against the fairness principle of the general-purpose operating systems and it may cause degradation in the overall processing performance of the system. There are several scheduling algorithms available for RTOS such as Liu and Layland Rate-Monotonic scheduling algorithm and Earliest Deadline Priority (EDF) algorithm [22]. Rate-monotonic scheduling is the common choice for preemptive scheduling. This scheduling requires processes to have static priorities based on the execution length. EDF algorithm dynamically assigns process priorities according to the deadlines.

Context Switching: It refers to the series of actions that happen when the computational resources transfer from Running process to the one in the Ready List. As the Running process changes its state, it immediately saves its registers to the TCB. It allows the process to restore its former execution state when it owns CPU once again. Context switching time is an important concern for RTOS as it happens more than often. This timeout depends on operating system TCB data structure as well as the processor architecture.

Interrupt Handling: Interrupt Request (IRQ) notifies the operating system for occurrences of external events. If the interrupts are not disabled, CPU polls for IRQ at the end of each instruction. When there is an IRQ available, processor terminates its current execution and identifies received interrupt signal. Then processor starts to service the interrupt. Operating systems treats incoming interrupts with a corresponding routine called Interrupt Service Routine (ISR). IRQ has the higher priority than the highest priority process. Therefore RTOS has to make ISR as short as possible and return back to the execution of processes. Interrupt latency is the elapsed time after receiving an interrupt signal until the start of the corresponding ISR.

Interprocess Communication: Processes running on the multitasking system may need to communicate with each other. Data structures within the operating system should be made available to only one process at any time. Otherwise, processes may end up using them inconsistently. Operating system maintains the consistency by allowing just one process to reach the data. Disabling interrupts temporarily or using binary semaphores are two possible approaches for the resolution. In the first method, a process disables all system interrupts during its execution. Therefore, it can safely reach the data structure without as nothing can interfere

its execution. One should never disable interrupts (even temporarily) without considering the interrupt latency of the RTOS. Long execution times with the masked interrupts will add-up to the worst-case latency because RTOS looses its control over the processes. It is usage is thus limited to tasks that have few instructions to complete. Using binary semaphore namely mutex is more secure but costly alternative. When a process needs resources in the critical section it informs the RTOS. Operating system will continue to have the control while processes are in the critical section, which means that RTOS may continue to provide real-time behavior. As the processes notify the RTOS, it takes more processing power to retain the critical sections in this way. While using mutexes, lower priority processes may lock the resources and higher priority process that needs those resources will not be able to continue its execution. In this situation, when a medium priority task starts it will preempt low priority task. In the end, medium priority task will run before the high priority task. This situation is called priority-inversion and should be avoided in RTOS. It can be accomplished with priority-inheritance; such that, lower priority process assumes the priority of higher priority process and continues its execution. Whenever it releases resources of the higher priority process it restores its original priority level.

Memory Management: Operating systems provide memory management service to control the portions of physical memory required by the processes. Memory management unit allocates programs requested memory and frees up unused portions. General-purpose operating systems provide the separation of process memory blocks by using virtual memory. Virtual memory is an abstraction layer between the physical memory and the process. It maps data structure used by processes to the locations in the physical memory. A process reaches its data through virtual memory. This isolation allows memory protection. Virtual memory has a demand paging utility. Demand paging dynamically exchanges the memory blocks between physical memory and the main storage unit. It stores the recently used memory blocks on the physical memory, whereas the formerly used portions are transferred to the storage unit. This procedure will suspend the process for an indefinite amount of time. Although virtual memory builds the proper abstraction to achieve memory security, it should be not be preferred in RTOS to avoid I/O latencies.

5.2 Measuring the Real-Time Performance:

In real-time multi tasking systems, several concurrent processes run in the same context. As the computational load on the operating system increases, it becomes harder to meet the required deadlines. A non-real time operating system may serve applications faster under idle condition; yet, RTOS continues to satisfy the requirements even if the system load increases. Thus it is not a practical way to obtain measurements under no load situations for a real-time behavior. While doing experiments on real-time processes, we applied computational load on the system by building a kernel configuration. The most important aspect of a real-time system is its predictability not the performance. Real-time system design should address worst-case situations and careful insight should be made for those assumptions. There is more than single method to measure the real-time performance. *Interrupt latency* and *context switching* time are the most common ones. In our implementation, the communication between different layers of protocols are provided by sending and receiving software interrupts. The receiver process will not wake up until it receives an interrupt from the sender process; therefore our measurement will include the sum of *interrupt latency* and the *context switching* time. We discuss communication latencies between processes for both: RTOS and non RTOS in Chapter 9.

5.3 Realtime Linux Operating System

5.3.1 Basics Of Linux Operating System

Linux operating system is free and open-source software that is widely used today in many platforms including personal computers, servers, mobile devices, and game consoles. It is distributed with GNU General public license that makes it so popular and is the main reason behind its rapid evolution in the past decades. GNU General Public License allows Linux source code to be freely used, modified and distributed with anyone either commercial or not [25]. Like in many modern operating systems, Linux separates virtual memory into Kernel Space and User Space. Core operating system functionalities are found in the Linux Kernel. It consists of a task manager, a file system manager, and the hardware access manager [26]. Some drivers for the hardware components are built into the kernel while others reside as Loadable Kernel Modules (LKM). LKMs extend base kernel when the corresponding func-

tionality is required at runtime. Through the insertion and removal of those modules, efficient usage of memory is achieved. User Space hosts to all of the software that is necessary for better user experience. User Space programs interact with the kernel space when needed. The separation of Kernel Space and User Space provides the efficient and safe use of resources.

5.3.2 Realtime Linux Kernel

Standard Linux Kernel is not suitable for real-time operation because it is designed to improve the common case performance. Realtime Linux kernel has a modified scheduler that supports Preemption. Existence of Big Kernel Lock (BKL) in Linux Kernel avoids Preemption of processes at the kernel space. In Realtime Linux Kernel, BKL is eliminated in all possible places. In addition, high-resolution timers are utilized in the Realtime Kernel to have system timer accuracy under 1μs. One of the biggest advantages of using Realtime Linux is, there is no need to use a specialized API for real-time processes. Ingo Molnar and Thomas Gleixner have great contribution for the existence of Realtime Linux. Currently, Open Source Automation Development Lab (OSADL) supports Realtime Linux; they organize conferences, measure and monitor the performance of latest kernel with a variety of hardware architectures and assist open source developers online [24].

5.3.3 Configure and Build Realtime Linux Kernel

Realtime Kernels emerge from some mainline Linux Kernels. Realtime Linux kernel is handled by patching the corresponding mainline kernel with the real-time patch. The latest mainline kernel for which the real-time patches are being developed is 2.6.33.7, though currently 2.6.39.2 is available. The latest stable kernel and the RT-Preempt patches are available online [27],[28]. After downloading kernel and its real-time patch, the user should extract and patch the mainline kernel with the following commands.

```
cd /usr/src/kernels
wget www.kernel.org/pub/linux/kernel/v2.6/linux 2.6.33.7.tar.bz2
tar -jxf linux-2.6.33.7.tar.bz2
mv linux 2.6.33.7  linux 2.6.33.7.2-rt30
cd linux 2.6.33.7.2-rt30
wget www.kernel.org/pub/linux/kernel/projects/rt/older/patch 2.6.33.7.2-rt30.bz2
bzip2 -d patch-2.6.33.7.2-rt30.bz2
patch -p1  patch-2.6.33.7.2-rt30
```

Before building real-time patched kernel, several changes should be made to the kernel configuration file. The user should open the configuration file and make the following changes.

```
make menuconfig
```

Enable HPET Timer Support and select Preemption Mode as Complete Preemption (Real-Time) shown in Figure 5.3. Disable tickless kernel and enable the High Resolution Timers as in Figure 5.4.

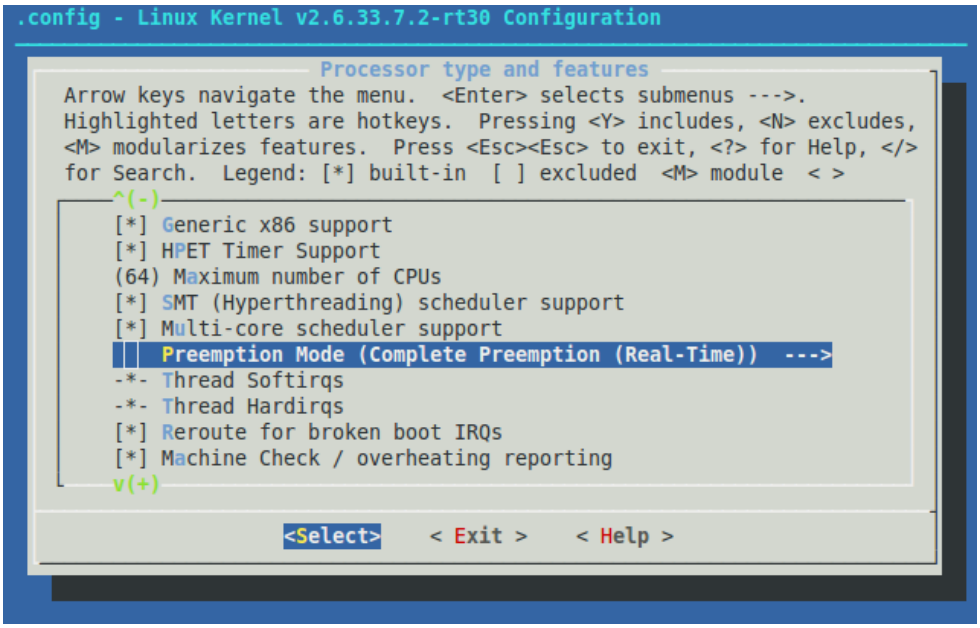

Figure 5.3: Configuration I

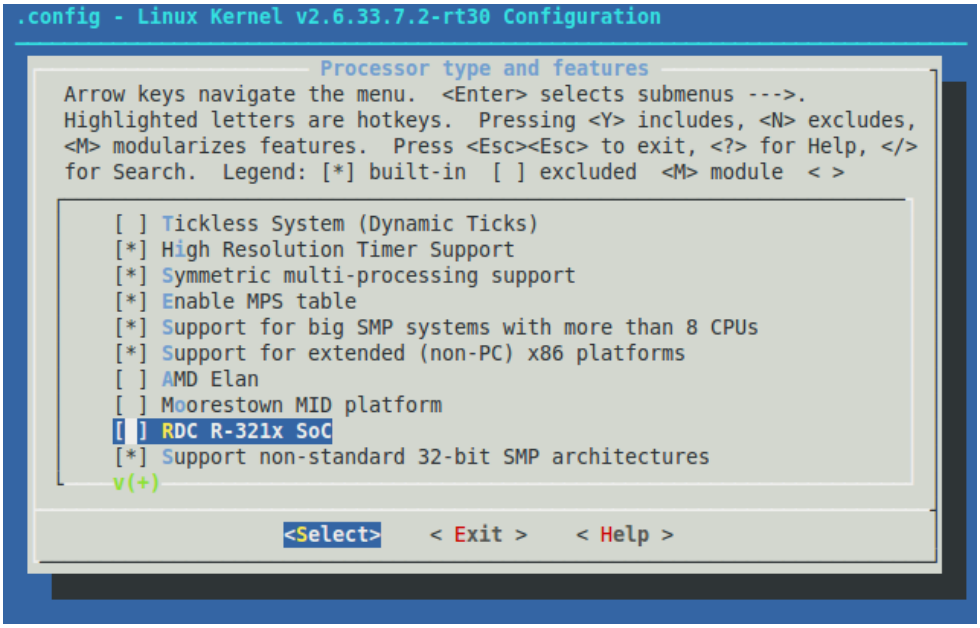

Figure 5.4: Configuration II

The new kernel is compiled, linked and installed as follows:

```
make
make modules_install install
```

Then linux bootloader (GRUB) should be updated to recognize the new kernel while booting up.

```
update grub
```

5.3.4 Programming with Realtime Linux

POSIX.1b API of Realtime Linux includes real-time extensions that provides Priority scheduling, Real-time signals, Clocks and Timers, Interprocess Communication and so on. In order to use these functionalities, programmers should link their projects with the real-time library using the option -lrt.

There are two critical things to do while programming real-time processes with Realtime Linux. Firstly, paging should be eliminated with the command `mlockall` to eliminate latency

while accessing secondary storage unit as described in Chapter 5.1. MCL_FUTURE option provides dynamic memory allocation at runtime will also reside on RAM and notifies if there is no space available. In addition to disabling paging, we should set priorities of the processes with sched_setscheduler command. Lowest priority process has a priority of 1 and for the highest process it is 99.

```
int main(int argc, char **argv) {

  mlockall(MCL_CURRENT   MCL_FUTURE); /*Lock process memory pages in RAM*/

  struct sched_param scheduling_parameters;
  scheduling_parameters.sched_priority = sched_get_priority_max(SCHED_FIFO);
  sched_setscheduler(0, SCHED_FIFO,  scheduling_parameters);

  //user code goes here...

  munlockall(); / unlock process memory pages in RAM /
}
```

We included above statements while implementing IEEE 1588 time synchronization protocol(Chapter 6) and the coordination layer protocol (Chapter 7). All threads of the implemented modules will be running with the highest available priority level which is 99. As the protocol implementation has to run on generic platforms, the code was written using the realtime extension of POSIX API.

CHAPTER 6

TIME SYNCHRONIZATION PROTOCOL IMPLEMENTATION

Our implementation of real-time industrial communication protocol relies on a synchronous time-slotting mechanism for medium access to achieve collision avoidance and avoid the non deterministic CSMA/CD behavior of IEEE 802.3. We employ a widely used IEEE 1588 protocol for the time synchronization of the nodes . It can be ported to any system in the IEEE 802.3 network topology. This portability provides us the ability to include synchronization software in any node without much effort. This section first compares and contrasts available time synchronization techniques to reason why we have chosen the IEEE 1588 protocol. Then we discuss several possible system clock types to choose the most suitable one during synchronization. Finally, we give the implementation details and the experimental results for the IEEE 1588 protocol.

6.1 Available Time Synchronization Techniques

There are several available time synchronization methods for different purposes. Figure 6.1 illustrates available synchronization methods and their inherent characteristics. To begin with, GPS devices provides time synchronization accuracy less than a microsecond. GPS receiver gathers clock information from the extremely precise atomic clocks. It provides the clock information to outside world with a Pulse Per Second (PPS) signal with a dedicated output pin. PPS signal creates an impulse with a duration of typically several microseconds at each second. A device that needs to be synchronized with a GPS time should process the PPS signal and adjust its local clock accordingly. A problem with the synchronization along with

a GPS receiver is, it imposes hardware dependability. In addition, these devices may fail to provide target accuracy at indoor operations.

Network Time Protocol is a standardized (RFC 2030) way of time synchronization applicable to devices with an internet connection. A local host updates its clock after receiving a NTP packet over the network. This packet contains the time server clock information along with some other controlling data. Considering the unpredictability of the latency between the time server and the local host, NTP may only provide synchronization accuracy in the order of hundred milliseconds. NTP is widely utilized on legacy data networks but our system needs much higher accuracy.

There are a number of synchronization methods that are using a propriety buses for cyclic communication. Sercos (IEC 61491) is an example to such synchronization technique. Proper clock source in this protocol is the decided master node. It sends time synchronization message with some millisecond intervals. Usage of separate bus for the synchronization increases the cost and expanding the system might not be easy.

Precise Time Protocol (PTP) is aimed to provide time-synchronization for the systems of local area network. IEEE defined the initial version of PTP in IEEE 1588-2002. Second version of PTP is announced in IEEE 1588-2008 standard. The revised version improves accuracy and performance especially when used with compatible network equipments. PTP is aimed to be used in relatively localized systems that may be of industrial automation or test and measurement environments. Although the proposed communication medium for PTP is Ethernet, it is also possible to run the protocol on any network that support multicast communications. PTP runs on the local area without a need for an administration. It decides the clock hierarchy during operation, therefore adding or removing nodes will not need a reconfiguration. Including an additional hardware for the NIC is proposed for higher synchronization accuracy (ie. <100ns). Software only implementation is an inexpensive and practical alternative that guarantees time synchronization in the order of microseconds. The prominent advantage of PTP is, it requires a little computational footprint. Therefore it is even possible to run this protocol on low-end embedded platforms. Running the software on a 66MHz m68 processor causes a CPU utilization less than 1% [10]. Furthermore, it imposes a little load to the network traffic during operation. Application of IEEE 1588 is announced by several organizations dealing with Real-time Ethernet concepts, such as: ETHERNET Powerlink, EtherCAT, CIPSync and

Profinet.

	IEEE-1588	NTP	GPS	SERCOS
Communication	Network	Internet	Satellite	Bus
Sync Accuracy	µs	ms	µs	µs
Hardware Dependant	For highest accuracy	No	GPS receiver and processor	Yes
Update interval	2 seconds	Several seconds	1 second	Each TDMA cycle
Application Area	Few Subnets	Wide Area	Wide Area	Few Subnets

Figure 6.1: Comparison of Time Synchronization Protocols

It is also possible to implement a NTP server within a local area network without an internet connection. This will result time-synchronization performance similar to that of the software only implementation of IEEE 1588. However, we preferred to implement IEEE 1588 protocol in order to be able to include hardware assisted 1588 network interface cards in the future.

6.2 System Clock

System clock serves the operating system and applications to indicate the time instant and generating timer events. System clock is critical while dealing with distributed real-time applications. Time synchronization performance is directly related to the system clock accuracy. PTP application should use a proper system clock to gather accurate timestamps. The same application makes corrections on system clock of slave nodes by resetting the clock to a new value and/or adjusting the frequency of system clock. Therefore, system timer should have a support for both of these operations. Precision of the system clock depends on the hardware clock source and the driver that lets user space to reach it. Operating system may use one or more of the available hardware clock sources.

6.2.1 Hardware Clock Sources

There are several alternatives that might be chosen as clock sources. A clock source often contains a hardware oscillator and a counter.

RTC (Real Time Clock)

Real time clocks are battery-powered chips that maintain the current time even if the related

device is off. They can only provide a clock resolution up to few milliseconds. Furthermore it takes quite a bit of I/O to read the clock. In modern computers, their main usage is for determining the system time on the startup and storing the current clock value just before shut down.

HPET (High Precision Event Timer)

High precision event timers are found on recent chipsets and they provide 10MHz 64-bit counters. It provides clock through an internal oscillator with a resolution of 1ns. The internal oscillator quality determines the accuracy of the clock. Variations in environment conditions, specifically the temperature, may cause intrinsic frequency instability in poor oscillators [31]. Non-deterministic clock drift will be observed under such conditions. Cost for reading a time from user-space application is around 1μs. HPET has been available for Linux operating system for 2.6 kernel versions with a POSIX interface.

Time Stamp Counter (TSC)

Time stamp counter is a CPU register already available in the widely-used Intel Pentium processor. It has a 64-bit cycle counter letting it to offer nanoseconds accuracy. Contrary to HPET which uses an internal oscillator, TSC increments the counter register at each processor cycle. Its time resolution depends on the processor speed; for high speed CPUs it may yield higher resolution than the HPET. For instance, a processor that is running at 2.2GHz will have a resolution of $\frac{1}{2.2GHz} = 0.45ns$. Furthermore, the time required to access the TSC register is much less than that of reading the HPET clock. `RDTSC` instruction reads the TSC register usually less than 50 CPU cycles. However, there are some reliability issues when using TSC. Multi-core CPUs contain a TSC registers corresponding to each of its cores and each may hold different values. In addition, power saving features of processor may cause CPU frequency scaling. As the CPU frequency changes, so does the increment rate of the TSC register. Lastly the portability is a problem with the TSC approach; as it relies on a specific processor registers.

Using TSC as a clock source instead of a HPET is shown to have a better synchronization accuracy [32]. Keeping the problems (especially, portability problem) associated with the TSC in mind, we use HPET in clock synchronization. By making this decision, we admit some degradation in synchronization accuracy. Nonetheless, we will be able to port our work without being constrained to a single processor family. Diminishing the frequency instability

in HPET oscillators by using better internal oscillators will provide better synchronization accuracy.

6.2.2 User Space Access to Clock Source

The standard Linux timers measure the time by just looking at the jiffies, which is the duration of the one tick in the system interrupt. HZ variable, which is contained in the kernel configuration file, lets the user to specify frequency of the system tick at the build time. Possible values for HZ are 100Hz, 250Hz and 1000Hz. It can be set at most 1000Hz that yields a tick resolution of 1 ms. This value is insufficient for most of the real-time applications. In order to provide more accurate time measurement, High Resolution Timers (HRT) are proposed. HRT eliminates the tick dependency (jiffies) of the time-keeping. The HRT framework has been introduced to the Linux system with the 2.6.21 kernel. HRT has accuracy up to 1ns depending on the timer hardware which is discussed in the previous part. It can be activated after setting the line CONFIG_HIGH_RES_TIMERS=y for configuration file in the kernel source directory before building the kernel. Then the High Resolution Timer API will be available for the user space through the POSIX interface.

POSIX.1b library defines real-time clock methods for `clock_gettime(clockid_t clk_id, struct timespec *tp)` and `clock_settime(clockid_t clk_id, const struct timespec *tp)` for retreiving current time and resetting it to a new value respectively. These methods are defined in the library time.h which should be linked against `librt` (-lrt option). Unix time value is represented in `timespec` structure, that is defined as:

```
struct timespec {
  time_t   tv_sec; /* seconds */
  long tv_nsec; /* nanoseconds */
};
```

Clock interface functions take `clockid_t` as parameter which may either be CLOCK_MONOTONIC or CLOCK_REALTIME. CLOCK_MONOTONIC provides frequency-stable monotonically increasing counter. It is not affected by the changes made to the system clock; it starts as the system boots and runs straight indefinitely. CLOCK_REALTIME represents basically the wall-clock of the system. Although CLOCK_MONOTONIC often provides better accuracy,

we use CLOCK_REALTIME option as we adjust the system clocks of the slave nodes while synchronizing with the clock source.

Our protocol will run on different systems and each them may have systems clock with different oscillator characteristics. IEEE 1588 protocol makes adjustments to the frequency of system clock on slave nodes to match the frequency of the master nodes oscillator. We use `adjtimex(struct timex *buf)` function to change the frequency of a system clock.

6.3 IEEE 1588 Time Synchronization Protocol

Our implementation of relative clock synchronization will be based on PTP version 2 (IEEE1588-2008). Relative clock synchronization focuses on time difference between nodes in local area network whereas in global synchronization, the aim is to achieve minimum offset from the absolute time. In the latter case there is a need for a source of a absolute time that can be provided by a GPS receiver. However, we will not need absolute time synchronization as our overall system will be closed loop and does not need an absolute time reference. PTP version 2 introduces the concept of *transparent clock* which is associated with the IEEE1588-2008 compatible network equipments. Transparent clock modifies the timestamp values in PTP messages according to the time spent while passing through the network device. Using a transparent clocks requires three additional protocol messages:`PDelay_Req`, `PDelay_Resp`, `PDelay_RespFollowUp`. We will not be using IEEE1588-2008 compatible network equipments, therefore we will not need those protocol messages.

We implement the following IEEE1588-2008 protocol messages.

- `Announce`
- `Sync`
- `Follow_Up`
- `Delay_Req`
- `Delay_Resp`

`Announce` messages are send by each node along with sending nodes' clock properties. IEEE 1588 defines the following properties of clock: *stratum number*, *priority* and *identifier*. Best

master clock (BMC) algorithm determines the most accurate clock among connected nodes with this information. `Announce` messages drives BMC algorithm to make computations periodically on each node. It must produce consistent results in all nodes. In other words, every node has to decide the same node as the master node after executing the algorithm. Master node will be a reference clock for slave nodes to synchronize their system time. PTP is self-organizing such that, BMC algorithm may determine a new master node when the former clock source leaves the network or a superior clock introduces to the network.

After settling on a clock hierchary, clock consumers synchronize with the clock source with the `Sync`, `Follow_Up`, `Delay_Req` and `Delay_Resp` messages. Figure 6.2 shows the exchange of protocol messages between two connected hosts.

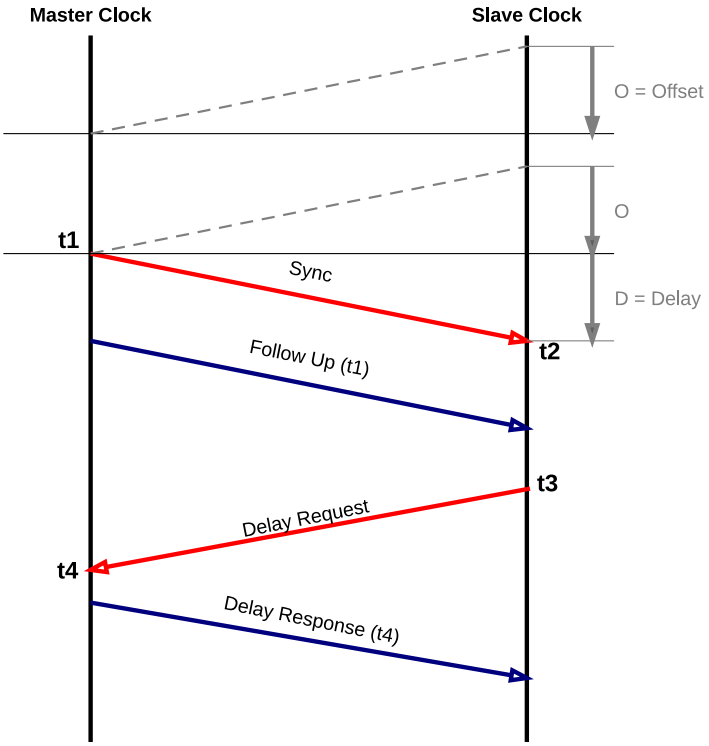

Figure 6.2: PTP Messages

PTP master multicasts `Sync` message typically in each second. It is followed by `Follow_Up` message that contains timestamp (t1) taken by master node while sending `Sync` message. Slave nodes will take a timestamp (t2) at the instant they receive`Sync` message. After receiving a `Follow_Up`, masters' timestamp can be extracted. Consequently, slaves will be able to

calculate master to slave delay by taking the time difference.

$$t_{ms} = t_2 - t_1 \tag{6.1}$$

Slave nodes send `Delay_Req` messages to the master node on random intervals uniformly distributed between 2 and 30 `Sync` periods. The idea behind sending at random intervals is to avoid burst network traffic directed to the master node. Slave nodes take a timestamp (t3) while sending`Delay_Req` packets. Then master node takes timestamp (t4) on receiving `Delay_Req` and sends back this information in the `Delay_Resp` message to those slaves. Slave to master delay can be calculated after receiving `Delay_Resp` message.

$$t_{sm} = t_4 - t_3 \tag{6.2}$$

The following two relations assume that *one_way_delay* is symmetric between master and slave.

$$t_{ms} = one\_way\_delay + offset\_from\_master \tag{6.3}$$

$$t_{sm} = one\_way\_delay - offset\_from\_master \tag{6.4}$$

Using (6.3) and (6.4) slave node can determine *offset_from_master*. These data's should be used with a care while adjusting the system clock. Slave nodes update t_{ms} more frequently than the t_{sm}. Therefore we will assume that t_{sm} remains the same where t_{ms} may be updated several times. It may lead us to wrong clock adjustments if either of them is erroneous. We make corrections to the system clock through the clock servo unit proposed in [10]. Not only it decides how to make fine adjustments to the system clock; but also it detects and diminishes jumps in the feed.

6.3.1 Clock Servo

This module is not a part of the IEEE 1588 protocol specifications, yet it is essential while making fine adjustments to the system clock. `Follow_Up` and `Delay_Resp` messages dynamically update clock servo variables. We have to record previous readings and use them in the

future as there might be misleading results at times. Figure 6.3 depicts the clock servo that is utilized while synchronizing with the master node [10].

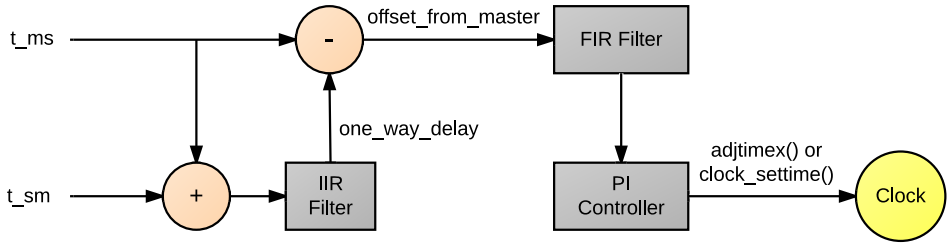

Figure 6.3: Clock Servo Diagram

`Follow_Up` messages trigger an update for the *offset_from_master*. Nodes take the average of previous two *offset_from_master* calculations to reduce the impact of unexpected jumps in the end. Actually, it is a basic two sample average FIR filter. We calculate the *one_way_delay* just after receiving the `Delay_Resp` packets. As the one way depends on many factors including the current network load, a more complex filter is introduced to predict the *one_way_delay*. It is a variable cutoff low-pass, infinite impulse response (IIR) filter that diminishes the short term fluctuations. Difference equation of the IIR filter is [10] :

$$s * y[n] - (s-1) * y[n-1] = \frac{x[n] + x[n-1]}{2} \tag{6.5}$$

In equation 6.5 x[n] and y[n] represents the most recent input and output of the filter whereas x[n-1] and y[n-1] are the previous values. In the same equation, variable s is stiffness and increasing it lowers the cut-off frequency of the filter. During the beginning of synchronization the value of s is set to be one and incremented gradually with each sample until it reaches to a predefined value.

Slaves make delay calculations less frequently than the offset calculations as they receive corresponding packets so. While calculating *offset_from_master*, we assume that *one_way_delay* stays the same till the next delay calculation. If a clock consumer node calculates an offset greater than 1 second, it will reset its clock with a new value calculated by subtracting offset from the current time. When the offset is less than a second, then slave node speeds up or slows down its oscillator to match the oscillator frequency of the clock source. In order to avoid aggressive changes to the clock frequency, we include a Proportional Integral (PI)

controller to mediate the clock frequency adjustment output. PI controller has the following difference equation [10] :

$$y[n] = \frac{y[n] - x[n]}{Ap} + a[n] \tag{6.6}$$

$$a[n] = \frac{y[n] - x[n]}{Ai} + a[n-1] \tag{6.7}$$

In equations 6.6 and 6.7 we represent recent filter input with x[n] and recent output as y[n]. The proportional gain (Ap) and integral gain (Ai) constants are used to determine the output characteristics of the controller. Default values for Ap and Ai are 1.

The clock servo that we use is described and its response is tested in a previous work [10]. It has been shown that employing clock servo with a maximum *stiffness* value of 6 ($s = 6$) attenuated most of the high frequency noise that causes unwanted impulses in the *offset_from_master* measurement.

6.3.2 Time-Stamping Mechanism

Timestamp accuracy for the PTP packets mostly determines synchronization accuracy. It is up to the requirements of the system to determine a suitable time-stamping method. Accuracy depends on the location where the timestamp has been taken in the network protocol stack. The error rate increases as the timestamp is taken at higher layers in the OSI model. Figure 6.4 illustrates possible locations for timestamp. There are several approaches for taking timestamp, though, highest accuracy can only be achieved with an additional hardware.

There are two alternatives for taking the timestamp if we are limited to the software only implementation. The first alternative will be taking the timestamp at the user-space simply after the socket receive or send routine returns. This approach will impose an aggregated latency and jitter of the underlying protocol stack. Superior timestamp accuracy is possible by taking timestamps at Network Driver level. In this method, timestamps for incoming packets are taken at the Network-Layer when the corresponding Interrupt Service Routine (ISR) is started for that packet. NIC modules provide receive ISR and in order to take a stamp in the beginning of the ISR one needs to modify the module and rebuild it against the kernel. Finally, the new module should be inserted to the system. It is not an elegant way to take timestamps with

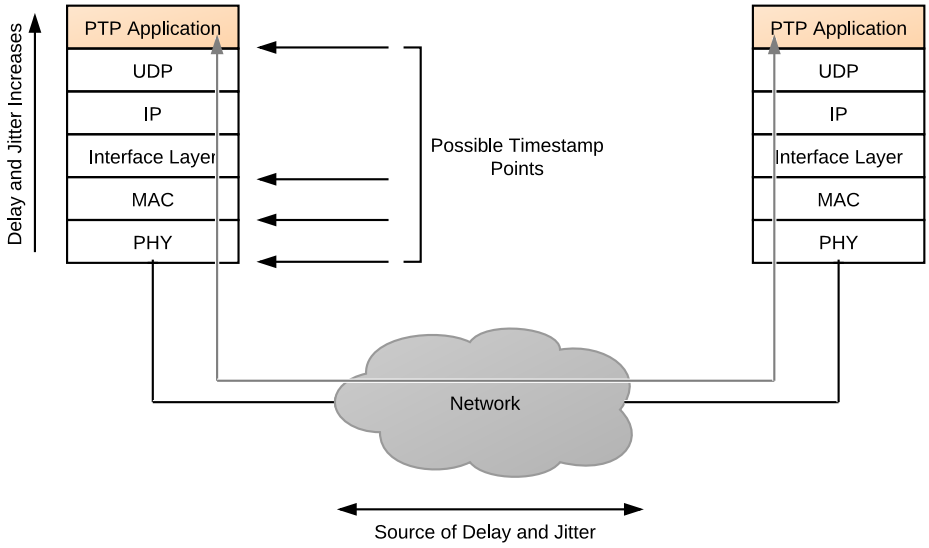

Figure 6.4: Possible Timestamp Locations

this method; as different manufacturers may have different drivers and kernel module programming needs lots of care. In addition, driver module should provide a separate interface to the user-space to deliver the timestamp values. Considering those problems, Linux kernel developers included SO_TIMESTAMP feature to the network stack. Linux mainline kernels of version greater than 2.6.30 have this feature. It allows user-space applications to access received timestamp information from the kernel without altering any driver modules. Timestamps for outgoing packets are taken by enabling loop-back by IP_MULTICAST_LOOP option and time-stamping the loop-back packets. Timestamp values are compensated by adding inbound and outbound latencies at the nodes. The main source of the error in this approach is caused by the operating systems' ISR latency.

Taking more accurate timestamps is only possible by including an additional hardware either at Media Independent Interface (MII), resides between MAC and PHY, or Physical layer. A time-stamp unit can be connected to the MII interface. This unit consists of FPGA and IEEE 1588 clock [29]. All packets traversing the IP stack also pass through this unit in parallel. Achievable synchronization accuracy is less than 100ns in a network of two hosts connected to a hub [30]. The second alternative is, using a PHY transceiver together with an IEEE 1588 clock. That newly developed transceiver allows taking timestamps with the lowest possible errors. The results have shown that 8ns synchronization accuracy is possible by using this chip

[33]. However in order to use DP83640, a suitable micro controller is needed. The problem in hardware modifications is that, timestamps are not taken with the system clock. System-clock should either be synchronized with IEEE 1588 clock or we have to use that clock instead of system-clock. PPS output signal of the IEEE 1588 clock can be used to synchronize system-clock. Using IEEE 1588 clock (monitor and adjust) in the application layer is possible by implementing a `ioctl()` command allowing clock operations .

Considering these methods, we decide to take timestamp from the NIC driver without making changes in the hardware. Accuracy of this method directly depends on the operating system behavior for the interrupts. Using a non-RTOS cannot avoid long durations of time when interrupts are disabled. Furthermore, bursty CPU or interrupt loads will cause delayed executions. These two factors contribute to the timestamp error as experimented in previous work [10]. As we are using Realtime Linux, the ISR latency will be limited. It will minimize the unexpected jumps for the execution of drivers' receive interrupt.

Synchronization protocol will run the on time-slotted medium which is provided by the Interface Layer of D^3RIP stack as illustrated in Figure 1.1. Synchronization packets are non-real time packets that are sent with the highest priority. Interface Layer may delay synchronization packets in the favor of real-time packets. However, D^3RIP protocol gives the real-time traffic higher priority than non-real time traffic. Hence, the outgoing packet timestamp values acquired from the network driver will not be accurate while running on top of the Interface Layer as the sending be delayed after taking the timestamp. Therefore IL informs PTP application to correct the acquired timestamps for sent packets by telling the time duration for synchronization packets waiting in the IL send queue. IL sends the delay of the sent packet after it is sent to network interface card driver then PTP uses this to have the correct time stamp (time of exiting the MAC driver). As the IL resides on the kernel-space this communication is provided by `ioctl` function.

There are two occasions that needs this correction. The IL in the master node corrects the `Sync` message timestamp before sending a `Follow_Up` message. After acquiring the correction from Interface layer, Master node sends the corrected `Sync` timestamp.

```
ioctl(rtOpts->ILDeviceFile, IOCTL_SYNC_CORRECTION,
  &ptpClock->syncCorrectionOffset.nanoseconds)
```

Similarly the ILs in the slave nodes correct the`Delay_Req` timestamp. This will increase the accuracy in *slave_to_master* calculations.

```
ioctl(rtOpts->ILDeviceFile, IOCTL_DELAYREQ_CORRECTION,
  &ptpClock->delayRequestCorrectionOffset.nanoseconds) > 0);
```

6.4 Time Synchronization Performance

In this part, we discuss experimental results while running software only implementation of IEEE 1588 version 2. Synchronization accuracy has a direct consequence in the D^3RIP protocol family. It determines the time-slot duration as we have to put *guard periods* within the time slots to compensate the amount of error in the synchronization of nodes. We will first analyze the effect of system clock and network equipments on synchronization performance. Then, we run time synchronization protocol on our target system. Because of the clock servo, it takes several minutes to synchronize with the clock source in the order of microseconds. In order to let the clock servo to settle down, we collected *offset_from_master* data five minutes after running PTP application.

6.4.1 Analyzing The Factors That Affect Synchronization Accuracy

We devote this section to reveal possible causes that may undermine synchronization accuracy. For simplicity, we run PTP application on two systems at a time. Each system operates on real-time patched linux kernel: Linux 2.6.33.7-rt30. The priority for the PTP application is set to be 99 which is the highest available real-time task priority. We configure `Sync` message period to be 1 second.

First we make a reference synchronization experiment consisting of two systems with the same manufacturers' system clock. We establish a direct connection with a crossover cable. After ten minutes of run, slave host's clock offset stabilized in the order of microseconds. The distribution of the offset from master node the related statistical data are illustrated in Figure 6.5 and Table 6.1.

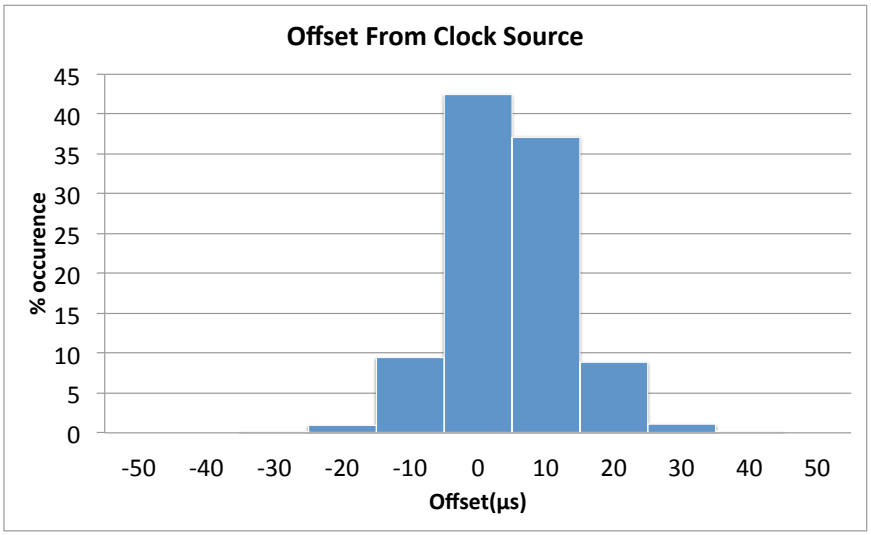

Figure 6.5: Synchronization Offset Distribution for Same System Clock Hosts, Direct Connection

Table 6.1: Synchronization Offset Values for Same System Clock Hosts, Direct Connection

Min:	-30.896μs
Max:	33.456μs
Average:	-0.2339μs
Deviation:	6.4053μs

Next, we connect the same PCs over Surecom 508T 10Mbps 10 port Ethernet Hub. To predict the influence of this Hub on the synchronization packets, we prepared a set-up with *Spirent AX4000 Network Analyzer* and simulated PTP traffic prior to the experiment. We produced a PTP traffic over the Hub by sending UDP packets (size=66 bytes) twice at a second. After running simulation for an hour we obtained the results for the specified Hub as seen in Table 6.2.

Table 6.2: Latency Caused by the Hub

Average Delay:	8.07us
Min Delay:	6.45us
Max Delay:	9.77us

Then we made an actual PTP synchronization with a duration of one hour over this Hub. The results are depicted in 6.6 and Table 6.3.

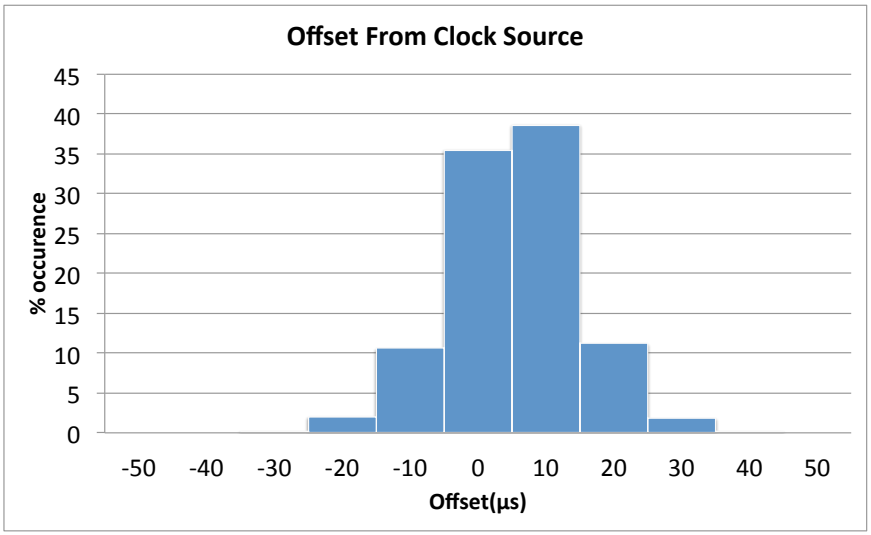

Figure 6.6: Synchronization Offset Distribution for Same System Clock Hosts, Connected over Hub

Table 6.3: Synchronization Offset Values for Same System Clock Hosts, Connected over Hub

Min:	-36.696μs
Max:	34.316μs
Average:	0.25151μs
Deviation:	7.1574μs

Lastly we run PTP application on hosts that have different manufacturers' system clocks. They are connected to each other with a cross-cable to isolate the latency caused by the Hub. Figure 6.7 and Table 6.4 shows the synchronization accuracy for this case.

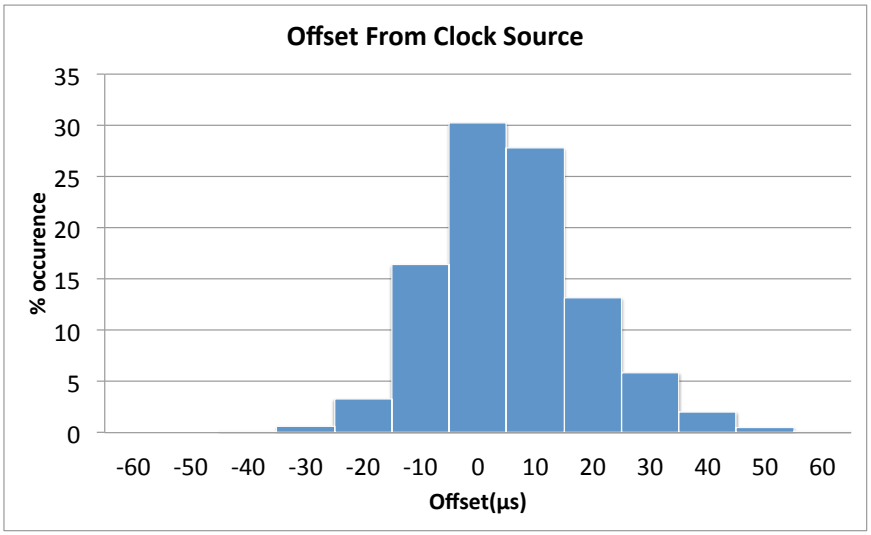

Figure 6.7: Synchronization Offset Distribution for Different System Clocks, Direct Connection

Table 6.4: Synchronization Offset Values for Different System Clocks, Direct Connection

Min:	-46.610μs
Max:	55.270μs
Average:	0.81231μs
Deviation:	10.389μs

Hosts that have the same system clock model have superior synchronization performance. The result is quite acceptable as the different oscillators have different drift rates. Including a 10 Mbps Hub instead of a direct connection between hosts, does not affect synchronization accuracy much. However in a network of burst network traffic, 10Mbps Hub may not be a sufficient and the synchronization accuracy will definitely decrease. For such cases, we should try using a faster Hub (eg.: 100Mbps), however these Hubs can be rarely found on the market currently.

6.4.2 Synchronization Performance for the Overall System

In this section, we give the results of the experiment that investigates the synchronization accuracy of our target system. We build a connection consisting of following nodes: 2 Desktop PCs, 1 Embedded Platform and 1 Industrial PC (IPC). They are all running the latest available realtime patched linux kernel version 2.6.33.7-rt30 and are connected to the network with the Hub. Duration of the experiment is 1 hour and we start collecting the synchronization error data after 5 minutes of the run. In this experiment, Best Master Clock algorithm of the PTP decided Industrial PC to be the master node.

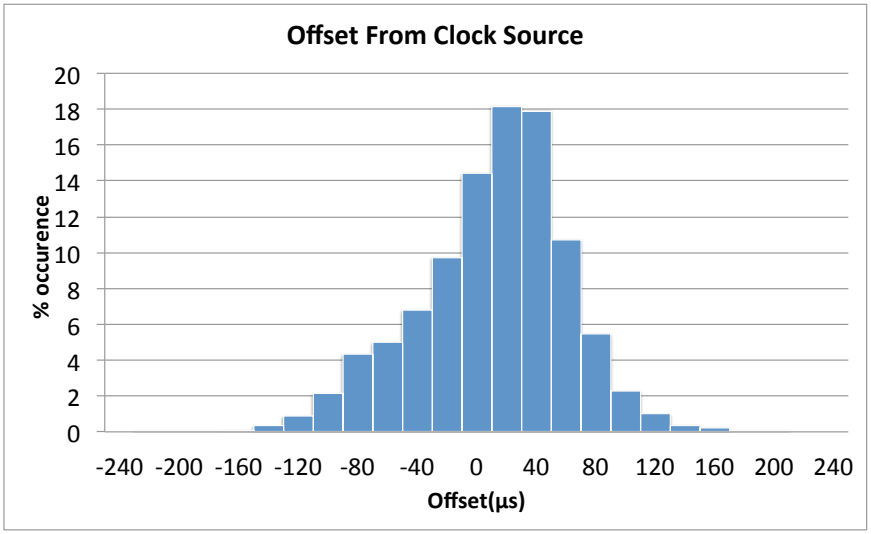

Figure 6.8: Synchronization Offset Distribution for PC1

Table 6.5: Synchronization Offset Values for PC1

Min:	-220.25μs
Max:	188.36μs
Average:	0.78271μs
Deviation:	39.403μs

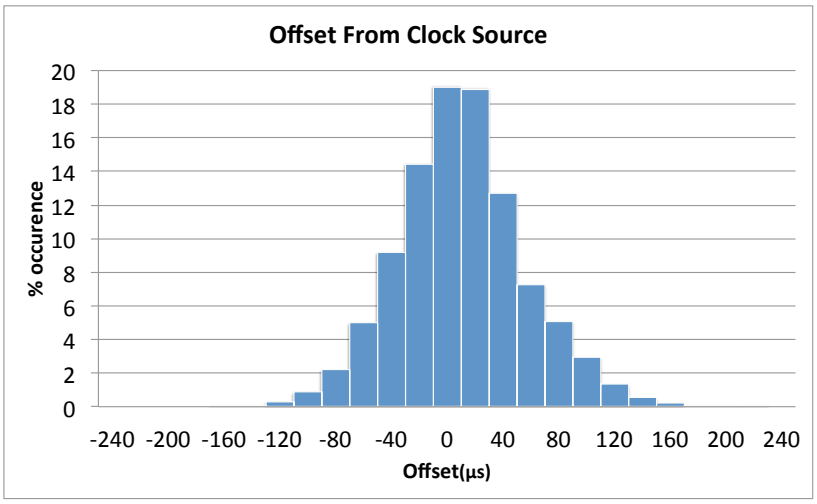

Figure 6.9: Synchronization Offset Distribution for PC2

Table 6.6: Synchronization Offset Values for PC2

Min:	-176.330μs
Max:	212.744μs
Average:	0.34973μs
Deviation:	35.169μs

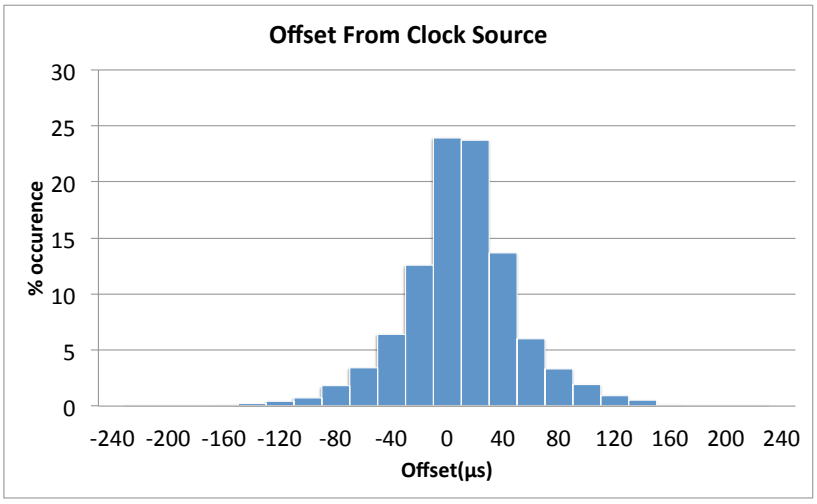

Figure 6.10: Synchronization Offset Distribution for Embedded Platform

Table 6.7: Synchronization Offset Values for Embedded Platform

Min:	-221.810μs
Max:	209.354μs
Average:	0.10815μs
Deviation:	30.251μs

It is seen that the synchronization performance is not as promising as the previous experiment that is consisting of just two nodes. Nevertheless, it is still in the order of hundreds of microseconds which can be tolerated with an appropriate slot duration selection.

Most of the available implementations and related experiments with the IEEE 1588 protocol in the literature are all concerned with the hardware assisted version of the protocol. It is reasonable to do so not only for its higher precision but also the measurements in the software only approach are not very accurate because it involves jitter related to the network stack. The recommended method for measuring the *offset_from_master* value is comparing the pulse per second output of the master clock to that of the slave node with an oscilloscope. We were unable to do this kind of measurement because we were limited with the HPET as a clock source which does not have a pulse per second output.

CHAPTER 7

COORDINATION LAYER PROTOCOL IMPLEMENTATION

Coordination layer (CL) assigns each time-slot, which is provided by the interface layer (IL), to a unique node with a distributed computation. In addition, it determines the type of the next slot which may either be RT (for Real-time traffic) Slot or nRT Slot (for non-real-time traffic). In the proposed protocol hierarchy, CL lies between the Control Application and IL. Essentially, CL processes RT messages that are coming from the Control application and performs a distributed calculation on the network to guarantee the delivery of the RT packets on time. CL adopts the slot allocation according the needs of the Control Application throughout the operation; as a result, the bandwidth is used more efficiently. CL protocol family consists of two different protocol: Dynamic Allocation Real-time protocol (DART) and Urgency-Based Real-time Protocol (URT) as discussed in sections 4.2.2 and 4.2.3.

7.1 Object Oriented Design for Coordination Layer

Our protocol family is developed through an object oriented approach. URT and DART extend a generic coordination layer protocol; yet, they use different methods for the decision of the next slot. Using a template and putting common properties together greatly enhances the code reuse. For example, if we propose a new CL Protocol in the future, then it will only require implementation of several functions while integrating it to the current architecture.

UML Class Diagrams [34] provide an elegant way to comprehend the structure of a system. Figure 7.1 illustrates the UML diagram for the coordination layer. `CLProtocol` class is an abstract class that controls the correct operation of the protocol and interfaces to other protocols. Internal variables for `CLProtocol` that change during runtime are accessed through the

class `rtOpts`. `DART` and `URT` derive from the `CLProtocol` by implementing the necessary functions. They have their decision variables as outlined in `vCLDART` and `vCLURT`.

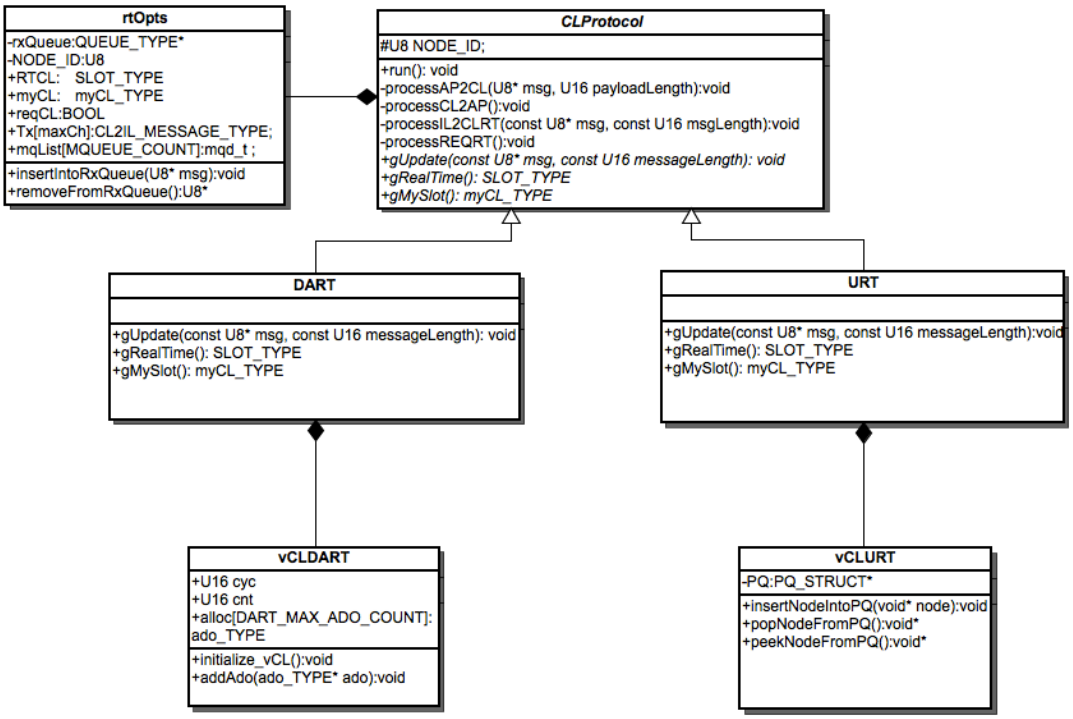

Figure 7.1: Coordination Layer UML Class Diagram

As the CL protocol has real-time constraints, our implementation:

- Uses no global variables at all
- Avoids redundant use of `memcpy` and `malloc`
- Avoids paging and runs with highest available real-time priority
- Utilizes POSIX API for its reliability

7.2 Generic Coordination Layer Model

In this part, common implementation that will be used by both DART and URT will be discussed. The implementation of so called `CLProtocol` class will serve a basis for the underlying protocols. It provides an ease for the further derivation of additional protocols besides DART and URT in the future. In addition, as this class will provide interfaces (Figure 7.2) to the other protocols in our framework, CL can adopt for any changes in other protocols by only doing some refinements to the `CLProtocol` class.

	Interfaces with Other Layers	
	Interface Layer	Control Application
Uses	CL2ILRT(RTCL, myCL,m)	-
Provides	REQRT(t), IL2CLRT(m)	AP2CL(ch,m) , CL2AP()

Figure 7.2: Interfaces Used and Provided by Coordination Layer

Figure 7.3 shows a sequence diagram for this class which a possible sequence of events that may happen at runtime according to the transitions as described in Chapter 4, Figure 4.1. Events coming from the Control Application are shown as (1), (5), (6). These events may occur anytime determined by Control Application. Event (1) and (5) poll CL for the availability of a new message from interface layer. Control Application sends (6) if its data is ready to be sent. CL receives IL2CL and REQRT events from lower layer protocol (2), (3), (7). CL should respond (3) and (7) with (4) and (8) respectively if and only if the elapsed time is less than *del* which is defined in Chapter 4.

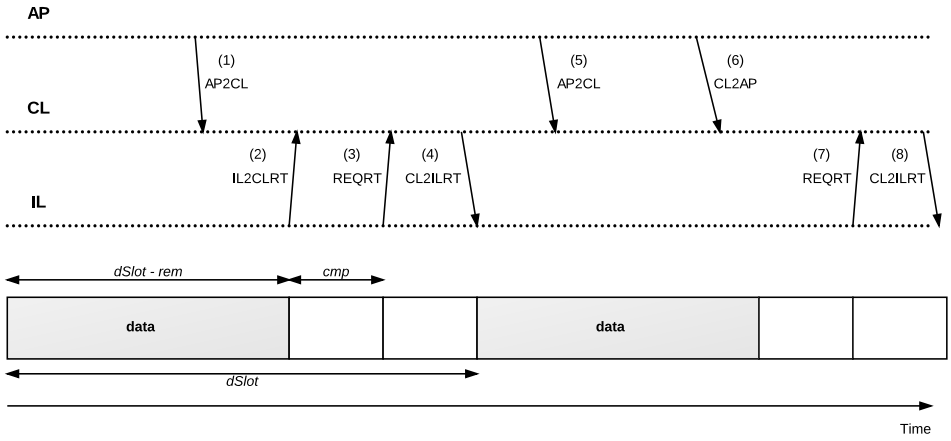

Figure 7.3: Coordination Layer Sequence Diagram

7.2.1 Communication with the Control Application

The transmissitted control message to the CL is regulated with a XML file in the control application which is further explained in Chapter 8. Control Application sends the prepared RT message to the CL by calling $\text{AP2CL}(dat, p, ch)_i$ along with the channel information and

the Control Message (Figure 7.4). It contains 1 Byte channel information, so that CL puts the received Control Message to the corresponding transmit buffer (i.e.: $\texttt{Tx}_i^{\text{d}}[ch]$). One thing to note here is that, CL Protocol Parameters change according to the underlying protocol; DART or URT. However, Generic CL model does not need to know the parameters at this time and puts them to the transmit buffer without extracting the CL Protocol Parameters.

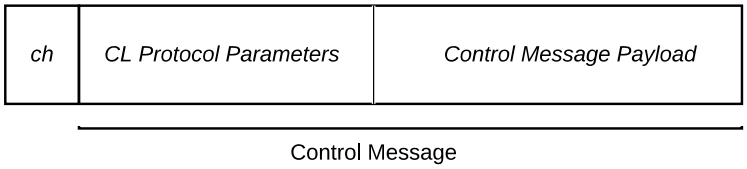

Figure 7.4: Control Application Message Contents

Control Application sends $\text{CL2AP}(\texttt{Rx}_i^{\text{d}})_i$ message to the CL in order to poll for any possible RT messages received from another node. CL receives Control Message of some other node from IL and puts them into the receive queue $\texttt{Rx}_i^{\text{d}}$. When CL receives $\text{CL2AP}(\texttt{Rx}_i^{\text{d}})_i$, it has two choices: either tell the Control Application that $\texttt{Rx}_i^{\text{d}}$ is empty or transmit the first Control message in the $\texttt{Rx}_i^{\text{d}}$. In both cases, a separate POSIX message queue provides the transportation of Control Messages between Control Application and the CL.

7.2.2 Communication with Interface Layer

IL sends $\text{REQRT}(t)_i$ message to the CL in order to retrieve the next slot information. This message includes a timestamp that represents the time message which left the IL. After receiving a $\text{REQRT}(t)_i$, Generic CL Protocol does the calculations for the next time slot with functions `gRealTime` and `gMySlot`. However Generic CL Protocol itself does not contain a definition for those methods. They are defined pure virtual functions in `CLProtocol` class as seen below.

```
virtual void gUpdate(const U8  msg, const U16 messageLength) = 0;
virtual SLOTTYPE gRealTime() =0;
virtual myCLTYPE gMySlot() =0;
```

Protocols that derive from Generic CL override those functions. Generic CL Model has two internal variables `RTCL` and `myCL`; they are updated with a `gRealTime` and `gMySlot` functions respectively. Finally, CL uses timestamp to determine whether the internal calculations has exceeded the time interval *del*.

CL protocol responds with CL2ILRT($\mathtt{RTCL}_i^{\mathrm{d}}, \mathtt{myCL}_i^{\mathrm{d}}, m)_i$ message after receiving a REQRT$(t)_i$. First two parameters are the updated internal variables. If the `myCL` variable signifies that the current node owns the next time slot, then m parameter will contain a Control Message stored in the corresponding $\mathtt{Tx}_i^{\mathrm{d}}[ch]$ buffer.

CL receives IL2CLRT$(m, t)_i$ message from IL that contains Control Message coming from some other node. After receiving a Control Message from IL, CL calls the `gUpdate` function with the Control Message in the parameter. As a result, the underlying protocol that overrides the `gUpdate` parses the CL Protocol Parameters and updates its current state.

Interface layer resides on the Kernel-Space; whereas, coordination layer is implemented on User-Space. Using a POSIX message queue in this case is not possible for this reason. Instead they share a character device and perform file operations on this device to communicate with each other. Interface layer implements the following file operations:

```
static struct file_operations fops = {
  .read = device_read,
  .write = device_write,
  .open = device_open,
  .ioctl = device_ioctl,
  .poll = device_poll,
  .release = device_release
};
```

CL opens a nonblocking character device and communicates with the IL by using the corresponding callbacks of the file operations.

7.3 DART

This protocol is aimed to allocate the necessary bandwidth for the RT traffic at the runtime. DART performs this task by dynamically mapping the time-slots to the RT packets of Control

Application. Decision variable for the bandwidth allocation is defined in vCLDART. Essentially, it consists of several Allocation Data Objects (ADO) that will share the available time-slots. ADOs are configured before the operation regarding the co-existing communication channels and the required deadlines. For example, if the system involves the communication of three different nodes at the same time, then number of ADOs should be at least three. In addition, each ADO owns number of time-slots to meet communication needs of the Control Application. This configuration is done during the initialization of the vCLDART as given below.

```
ado_TYPE ado[3];
ado[0].num = 2; ado[0].slots[0] = 0; ado[0].slots[1] = 3;
ado[1].num = 2; ado[1].slots[0] = 1; ado[1].slots[1] = 4;
ado[2].num = 1; ado[2].slots[0] = 2;
addAdo(&ado[0]);
addAdo(&ado[1]);
addAdo(&ado[2]);
```

Decision variable vCLDART stores these ADOs in the alloc array. When a node uses an ADO then, alloc.used will be its *Node Id* together with the *channel*. Assignments of ADO to a node happens within the gUpdate function of the DART. This function extracts the DART Parameters in the message (Figure 7.5) and updates alloc.used accordingly. DART parameters are composed of number of New Channel Requests (NoN), number of Free Channel Requests (NoF) and the request contents. New channel request tells the required number of slots to allocate for the particular channel of a node. Considering the required number of slots, gUpdate picks up the most suitable unused ADO and makes the assignment. Free channel request specifies that the given node no longer uses the channel. As a result, gUpdate revises the decision variable so that, the unused ADO will be available to other nodes.

vCLDart holds the current time-slot count in the *cnt* variable. It is incremented with modulo *cyc* whenever the interface layer issues a REQRT$(t)_i$ message. DART makes decisions about the next time-slot with its implementation of gRealTime and gMySlot. gRealTime returns true if the current slot (*cnt*) is used by any of the ADO. In the case that there is an alloc[i] that uses the current time-slot, then gMySlot looks at the alloc[i].used and returns true if it is equal to the current Node Id.

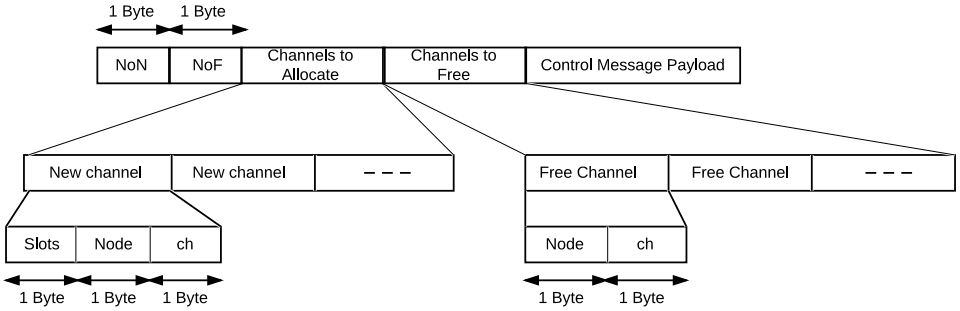

Figure 7.5: Contents of the DART Message

7.4 URT

The operation of URT protocol is determined by the priority queue that contains communications request. `vCLURT` initializes the priority queue and provides the wrapper function to add/remove elements. Initialization of the priority queue is important since the protocol may fail to operate if it is not made properly. The priority queue might be configured as below:

```
getTime(&curTime);
comReqPtr->nodeId = 1;
comReqPtr->ch = 1;
comReqPtr->initialeT = 0; //in ms
comReqPtr->initialdT = 10; //in ms
t.nanoseconds = (comReqPtr->initialeT)*1000000;
comReqPtr->eT = addTime( &curTime, &t );
t.nanoseconds = (comReqPtr->initialdT) 1000000;
comReqPtr->dT = addTime( &curTime, &t );
insertNodeIntoPQ(comReqPtr);
```

URT message format is depicted in the Figure 7.6. First byte in the message is the Number of Requests (NoR) field that carries the included request count information. Communication request is a tuple that holds node, channel, eligibility time and deadline(N, ch, eT, dT). Eligibility time and deadline values are relative and are expressed in milliseconds.

After receiving a URT message from IL, `gUpdate` function handles the update of the priority queue according the the requests in the CL Protocol Parameters field. Priority queue stores elements in the same order as communication requests. However, eT and dT values represent

the absolute times. Therefore before inserting an element into the priority queue, we add the current time to *eT* and *dT* enclosed in communication request. Priority queue orders communication requests according to their absolute deadlines. `gUpdate` function is responsible for removing the topmost communication request when a node owning that request successfully transmits its Control Message. It is also possible that node has no Control Message to transmit although it owns a communication request. It sends a dummy message with empty CL Protocol Parameters field to inform all other nodes that it has no message to send. In that case, `gUpdate` puts unused communication request into the priority queue once again with updated *eT* and *dT*.

`URT` decides on the next time-slot by inspecting the priority queue momentarily. If there is a communication request that has a eligibility time less than the current time, `gRealTime` returns with real-time slot. Distributed computation of `gMySlot` determines the node that owns the next real-time slot by looking at the Node Id in that communication request.

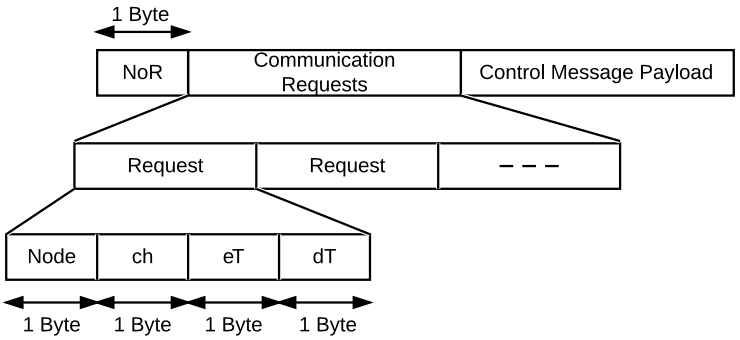

Figure 7.6: Contents of the URT Message

7.4.1 Priority Queue

Implementing a computationally efficient priority queue is essential for URT. This queue should order the communication requests according to their deadlines as discussed previously. Our implementation of priority queue will be based on binary heap structure. It sorts the elements according to the *key* values. In binary heap structure, every level is filled except for the lowest level. Each node may have two children nodes which strictly have a lower *key* than the parent node. The lowest level is composed of leave nodes which do not have any

children. If there is a tree of `n` nodes then, the depth of the binary heap will be `log(n)`.

Removing a Node: The root node of the binary heap structure has the highest valued *key* among the nodes. Therefore, the `Pop` command will just remove the root node and put the rightmost leaf to this place instead. Then, the new root node for the binary heap will be determined recursively comparing the *key* with the children node and doing a swap operation if the children has a higher *key*. The cost of determining the new root node will be proportional to the depth of the binary heap structure in the end.

Inserting a Node: `Push` command places the node in the argument to the lowest level in the binary heap structure as a leave node. Then a recursive comparison and swap operation is performed until its parent node has a higher *key* than the inserted one.

In both cases, the number of swap operations may not exceed the depth of the binary heap structure; thus the computational complexity of both insertion and removal is `O(log(n))`.

CHAPTER 8

INTEGRATING THE COMMUNICATION RELATED INFORMATION INTO THE CONTROL APPLICATION

In Chapter 3, we explained that jobs contain a communication requests enabling the transmission of subsequent jobs on our protocol. In this chapter, we will describe how to integrate the control application [2] with the coordination layer protocol. We continue with the simple machine example and describe the integration process for both members of coordination layer protocol.

8.1 Communication of Shared Events

Control application sends the available shared event(s) to the coordination layer together with the protocol parameters as given in Figure 7.4. The information about protocol parameters and the mapping that translates an event to a `EventId` is stored in the XML files created for each controller. As there might be several events that have to be sent in a single message, the first byte of the *control application payload* is occupied by *total number of events to send* value. Input and output events are all identified by 1 Byte `EventId` field which should be consistent in all XML files. While exchanging shared events, sender node transmits `EventId` and the receiving node identifies the corresponding event by looking up the local XML file. Therefore *total number of events to send* is followed by specified number of `EventIds`. Preperation of *CL Protocol Parameters* field depends on the underlying coordination layer protocol.

8.1.1 Integrating the Control Application With URT

While working with the URT, XML file contains all communication requests related to the output events. In this file, we can specify the communication requests in the format that URT needs as described in the Section 4.2.3. We can append bunch of communication requests to a single output event. Control application basically parses the events and the specified parameters for the URT. Whenever a shared event is to be send to another controller, it will set the *CL Protocol Parameter* field according to the specifications in the XML file. For the simple machine example, Controller G1 has a XML file that contains the event configuration given in Listing 8.1.

```
<?xml version="1.0" encoding="ISO-8859-1" standalone="no"?>
<!DOCTYPE D3ripURTDevice SYSTEM "d3ripURTdevice.dtd">
<D3ripURTDevice name="ControllerG1_Net">
<!-- Time scale in ms/ftiu -->
<TimeScale value="1000"/>
<EventConfiguration>
<Event name="?mue" iotype="input">
<EventId value="1"/>
</Event>
<Event name="!mue" iotype="output">
<EventId value="2"/>
<ChannelToTransmit value="1"/>
<ParameterRecord>
<DestinationNode value="2"/>
<DestinationChannel value="1"/>
<EligibilityTime value="2" />
<DeadlineTime value="5"/>
</ParameterRecord>
</Event>
<Event name="mue" iotype="input">
<EventId value="3"/>
</Event>
</EventConfiguration>
</D3ripURTDevice>
```

Listing 8.1: XML Event Configuration: Controller G1 for URT

8.1.2 Integrating the Control Application With DART

Similar to the previous section, we define XML files to extract communication needs of the control application in the format described in Section 4.2.2. Now the XML file contains the DART related parameters such as *channels to allocate* and *channels to free* belonged to the output events. Whenever control application sends an output event, it will include related protocol parameters in the *CL Protocol Parameter* field. Consequently, DART can progress after receiving control application messages. Considering simple machine example, XML file for Controller G1 should include the configuration in Listing 8.2.

```
<?xml version="1.0" encoding="ISO-8859-1" standalone="no"?>
<!DOCTYPE D3ripDARTDevice SYSTEM "d3ripDARTdevice.dtd">
<D3ripDARTDevice name="ControllerG1 Net">
<!-- Time scale in ms/ftiu -->
<TimeScale value="1000"/>
<EventConfiguration>
<Event name="?mue" iotype="input">
<EventId value="1"/>
</Event>
<Event name="!mue" iotype="output">
<EventId value="2"/>
<ChannelToTransmit value="1"/>
<ChannelsToAllocate name>
<DestinationNode value="2"/>
<RequiredSlots value="1"/>
<DestinationChannel value="1"/>
</ChannelsToAllocate>
<ChannelsToFree name>
<DestinationNode value="1"/>
<DestinationChannel value="1"/>
</ChannelsToFree>
</Event>
<Event name="mue" iotype="input">
<EventId value="3"/>
</Event>
</EventConfiguration>
</D3ripDARTDevice>
```

Listing 8.2: XML Event Configuration: Controller G1 for DART

CHAPTER 9

EXPERIMENTS AND RESULTS

This section is devoted to the experiments related to the D^3RIP [19] protocol family. In the first part, we examine communication latencies between the control application, coordination layer and interface layer running in a single platform. It is necessary to introduce such an experiment because coordination layer protocol has time constraints while responding control application and the interface layer. In addition, it will verify the reasoning behind using a real-time operating system instead of a generic operating system for D^3RIP. In the second part, we include experimental results for a complete distributed control system communicating with D^3RIP and IEEE 1588 time-synchronization protocol implementation presented in this thesis. The operation of the complete system will be described for each of element of coordination layer and interface layer protocol family including the time-synchronization and non real-time messages.

9.1 Interprocess Communication Latency Experiment

It is critical for coordination layer (CL) to meet several deadlines for the correct operation of the system. This section will discuss the communication latency of CL with other protocols in the framework. We run these tests on both a Realtime Linux and its non real-time counterpart. Testing platform for this section will be a PC with Intel Core i3 550@3.20GHz Processor and 4GB of RAM. Our experiment not only contrasts the real-time performance of RTOS and non-RTOS, but also it shows us the limits of Realtime Linux running on our system. We use the latest available stable Realtime Linux kernel 2.6.33.7-rt30. As a generic-usage linux distribution we will be using 2.6.35-generic kernel. We sample the maximum latency data for a period of 50ms and tabulate the results.

9.1.1 Communication Latency with Control Application Process

The communication between Control Application and CL is based on POSIX message queues. We have to determine the latency between writing to a message queue with a Control Application process and reading of that message by the CL. For this experiment, Control Application process generates a 200 Bytes long AP2CL message together with a *send timestamp* at each millisecond and sends it to the message queue. On the receiving side, CL waits for a Control Application message from the message queue. CL takes a *receive timestamp* at the instant it receives a message and extracts the *send timestamp* from message contents. CL calculates the time difference from these timestamp values to measure the communication latency in between. Figure 9.1 shows the communication latency when operating under generic-usage Linux whereas Figure 9.2 shows the same for Realtime Linux.

9.1.2 Communication Latency with Interface Layer

CL communicates with interface layer (IL) through a character device file. In this part, we test the communication latency that occurs while reading commands from the opened device file. For testing purposes, IL sends 200Bytes of message IL2CLRT message at each millisecond including a send timestamp in the payload part. CL receives message and calculates the latency by subtracting the timestamp from the instant it receives the message. Figure 9.3 and Figure 9.4 shows the communication latency for interface layer messages for daily-use Linux and Realtime Linux respectively.

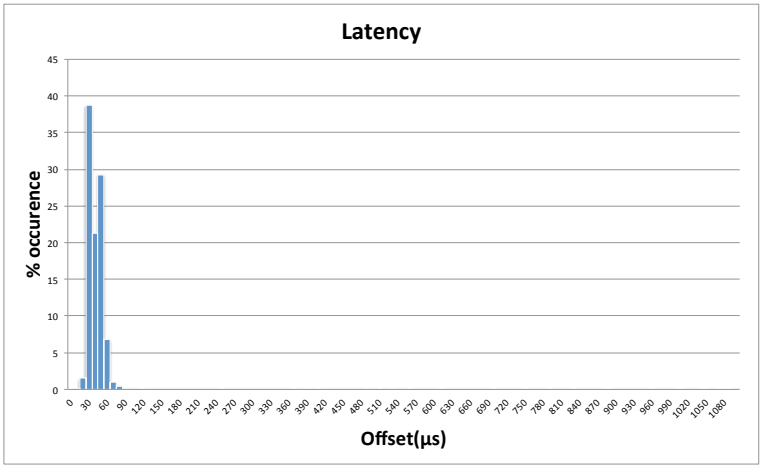

Figure 9.1: Communication Latency with Control Application using non-RTOS

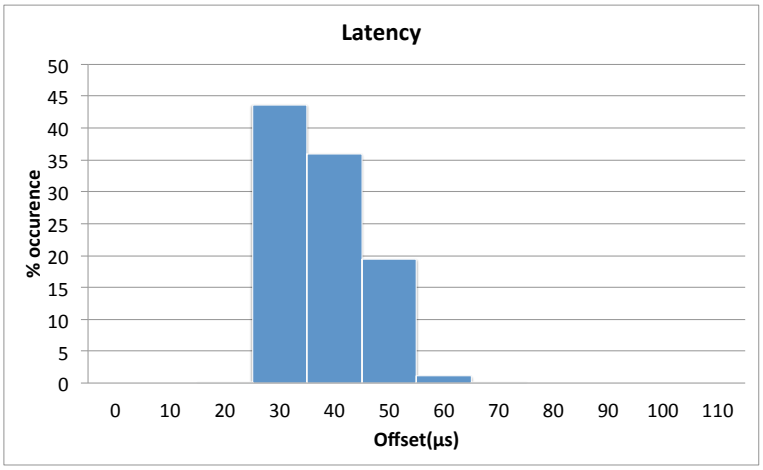

Figure 9.2: Communication Latency with Control Application using RTOS

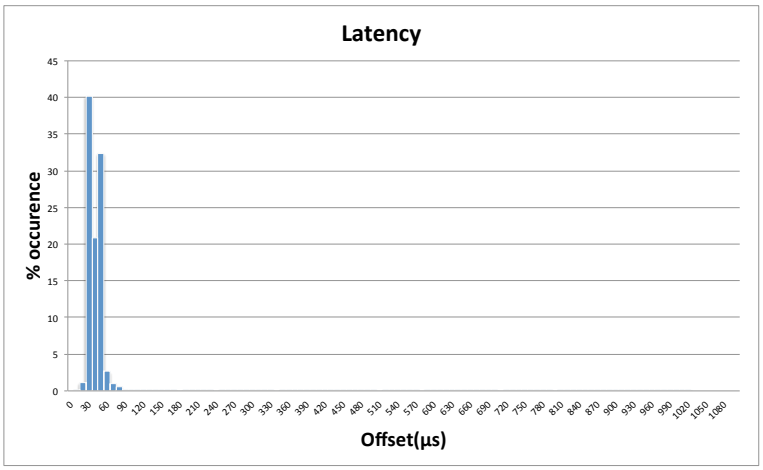

Figure 9.3: Communication Latency with Interface Layer using non-RTOS

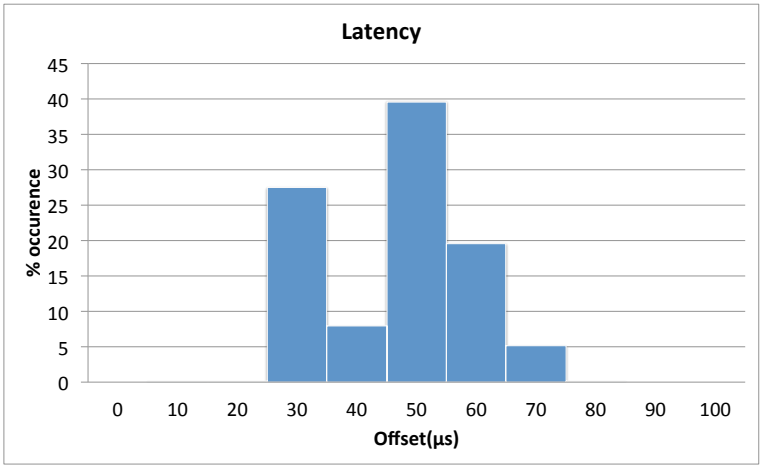

Figure 9.4: Communication Latency with Interface Layer using RTOS

These latency figures are important as the coordination layer has to reply REQRT message of the interface layer in a time bounded with $rem - cmp$ as given in Chapter 4. Ideally, there should not be any latency while communicating between threads of different processes. The latency here is composed of the time required to start an interrupt service routine for the software interrupt generated by the running process and context switching time to wake-up the sleeping process as we discuss in Chapter 5.

Using generic Linux kernel yields latencies over 1ms while reading from Control Application or interface layer. For the Realtime patched Linux kernel, the maximum latency is 70μs when reading from Control Application and 85μs for the interface layer. Slight increase in the latter case is caused by copying the buffer from the kernel space to the user space. Unless we used a RTOS, it would not be possible to achieve a bounded latency measurements in the order of microseconds. Results of this section directly reveals the real-time performance of the Realtime Linux running on our test platform. The performance of Realtime Linux may change as we deploy to another PC containing a different processor. Currently, an official web site for Realtime Linux continuously compares and contrasts timer interrupt latencies for the most recent version of real-time kernel on varying platforms [24]. Our results in this section are comparable to a similar system described in [24].

9.2 Complete Operation of the Distributed Controller System Experiment

In this section we further explain all the operational principles of the distributed controller system based on the example given in the Chapter 3. Then we give experimental results for URT protocol of coordination layer running top of the RAIL as an interface layer protocol.

9.2.1 Operation of The Distributed Control System Example

Distributed controller system example given in the Chapter 3 can be realized as given in Figure 9.5. It is seen that Controller G1 and G2 communicate with the plant over a serial line whereas the shared jobs between these controllers are sent over the Ethernet operated by D^3RIP.

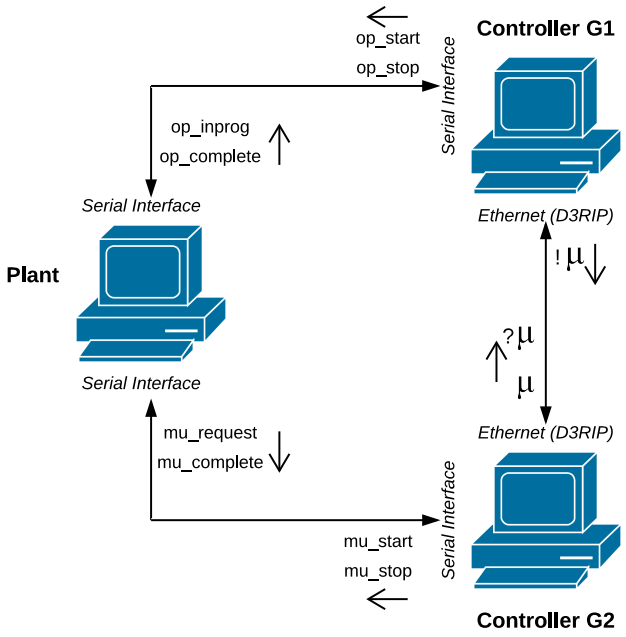

Figure 9.5: Experimental Setup

During the beginning of the operation we start the IEEE 1588 protocol on both of the controllers without initiating the time-slotted operation. As the controllers have their clocks synchronized in the order of microseconds we start the interface layer module to operate on the time-slotted medium. Interface layer is now ready to poll real-time packets coming from coordination layer and non real-time packets from other applications. Depending on the choice of interface layer protocol, the frequency that it polls the coordination layer changes.

Operation with RAIL: If the interface layer protocol is selected to be RAIL, then it will ask coordination layer for the existence of available real-time packets at several slots periodically depending on the `railILSchedule`. Current slot-type in the `railILSchedule` advances in the end of each time-slot. RAIL asks coordination layer protocol to send real-time packets when the current `railILSchedule` points to the *RTSlot*. If the current slot is a *myNRTSlot* then RAIL transmits the non-real time packets waiting in its queue. In such case, other nodes (in this example there is only one) should have *otherNRTSlot* in their current `railILSchedule`. The queue for the non real-time packets is a priority queue that contains the IEEE 1588 protocol packets with the highest priority. Therefore time-synchronization will not be interrupted because of the network traffic as IEEE 1588 packets are chosen to be sent before any other non real-time packets.

We configure time-slot schedule of RAIL at Controller G1 to be

```
const unsigned short int railILSchedule[RAIL_CYCLE] = {RTSlot,myNRTSlot,RTSlot,↩
    otherNRTSlot};
```

Controller G2 has the following schedule consistent with the G1

```
const unsigned short int railILSchedule[RAIL_CYCLE] = {RTSlot,otherNRTSlot,RTSlot,↩
    myNRTSlot};
```

Operation with TSIL: When the TSIL is chosen as an interface layer protocol, it inquires coordination layer in every time-slot for available real-time packets. Therefore, coordination layer protocol decides the next slot type to be either real-time or non real-time. If the next slot is determined to be non real-time slot, then TSIL lets only one node to transmit its message; this decision is made by looking at the `nRTSet` variable. Once again, the selected node transmits IEEE 1588 packets before any other non real-time packets.

In either case, coordination layer family should be ready to respond the coming requests from the interface layer at any time. Coordination layer protocol does not have a knowledge of the passing time until interface layer makes a request to it. For the simplicity of the example, we will use only one channel for coordination layer protocol. It has to be noted that, depending on the choice of coordination layer protocol, control application generates appropriate protocol parameters for URT and DART. Controller G1 is identified as NodeId=1 and for Controller G2, NodeId=2 by the coordination layer protocols.

Operation with URT: If we operate distributed control system with URT then we have to initialize its priority queue to contain a *communication request*: (2, 1, 2ms, 5ms) to let Controller G2 transmit the first shared job $?\mu$. Until URT receives the $?\mu$ job from control application, the communication request in the priority queue will re-enter the priority queue with the updated times triggered by the request from the underlying interface layer. When URT has $?\mu$ ready to send, it will be sent with the first request from the interface layer after checking that the *eligibility time* for the communication request is less than the current time. The $?\mu$ message will contain a request: (1, 1, 2ms, 5ms) to allow Controller G1 sending the reply with $!\mu$ message. Similarly $!\mu$ will contain a request: (2, 1, 2ms, 5ms) to allow the latest shared job μ to

be sent. Lastly, controller G2 sends μ with the request: (2, 1, 2ms, 5ms) and the transmission of these 3 shared jobs starts over from the beginning. For this example, note that size of the priority queue holding communication requests never exceeds one.

Operation with DART: We need to configure allocation data object of DART for this example as follows: alloc[1].num = 1 alloc[1].slots=[0]. Using only one allocation data object suffices for this case because there is at most one shared jobs to be transmitted at a time. In the beginning, we should initialize the DART to enable Controller G2 to send $?\mu$ by setting alloc[0].used = (2, 1). Now, the real-time bandwidth will be belonged to Controller G2 until control application sends $?\mu$ message to the DART. Then DART forwards $?\mu$ job upon a request from IL with the following protocol parameters: channel to allocate: (1,1,1) and channels to free: (2,1). These parameters allow Controller G1 to transmit $!\mu$ as the bandwidth allocation will be shifted to the Controller G1. Controller G1 transmits $!\mu$ together with channel to allocate: (2,1,1) and channels to free (1,1) when interface layers asks to do so. Finally, Controller G2 sends μ by including following DART protocol parameters channel to allocate: (2,1,1) and channels to free (1,1) to restart the communication of the jobs from the beginning.

9.2.2 Experimental Results for the Distributed Control System Example

In this section, we demonstrate the experimental results as we run URT protocol on top of the RAIL. Controller G1 and G2 are connected over a Hub as in Chapter 6.4.1. These controllers are identical PCs equipped with Intel Core i3 550@3.20GHz Processor and 4GB of RAM. We set $dSlot = 3ms$, $rem = 0.5ms$ and $cmp = 0.1ms$ which we decided according to the time synchronization performance in Chapter 6.4.1 and the communication latency presented in Chapter 9.1. We generated a non real-time traffic by sending 150Bytes dummy packets in each controller with a 10ms interval.

Measurements are taken at the control application by calculating the elapsed time between a packet is sent to coordination layer and the coordination layer sends it to the interface layer. The result of the observed latencies for a 24 hour experiment is as following:

Table 9.1: Latency Measurement for Real-Time Messages

Min:	486.8μs
Max:	6.573ms

The results are consistent with theoretical calculations. Maximum latency would expected to be: $rem + RTSlot + nRTSlots = 6.5ms$ in the case control application sends real-time message to the CL just after IL sends REQRT and the next slot is a RTSlot. The minimum latency can be $rem = 0.5ms$ if the next slot is a RTSlot and control application sends real-time message just before interface layer sends REQRT to the CL.

CHAPTER 10

CONCLUSION

The goal of this thesis is the implementation of the industrial real-time Ethernet protocol family D^{3}RIP that dynamically adapts to the communication needs of a distributed control application. The main tasks performed in this thesis are:

- Coordination layer
- IEEE 1588 synchronization
- Integration of the coordination layer and an interface layer that is implemented in the scope of another master thesis
- Integration of the coordination layer and the control application

In addition, the thesis conducts experiments in order to validate the Dynamic Distributed Dependable Real-Time Industrial communication Protocol (D^{3}RIP) [19] implementation.

Performance metric for the IEEE 1588 protocol was the *offset from master* value that stands for the time difference between the system clocks of clock source and other nodes. First we evaluated factors that might effect synchronization performance in Section 6.4.1. It is shown that, synchronization accuracy drops when the systems clocks of the hosts are not identical and it is verified that including Hubs do not have a significant affect on synchronization performance. Next, the synchronization accuracy that can be achieved by our target system is experimented in Section 6.4.2. Results of this experiment directed us to choose a suitable time-slot duration for TDMA to tolerate errors in the synchronization accuracy.

For the coordination layer protocol family, the communication latencies with interface layer and control application are measured. In Section 9.1.1 the communication latency between control application and and coordination layer is shown to be under 100μs provided that a real-time operating system is used. Similar experiment is made to measure the latency between coordination layer and interface layer in Section 9.1.2. Results were similar to the previous

experiment while running real-time operating system. These experiments have shown the capabilities of real-time operating systems as we run the same tests on non real-time operating system. It is shown that the Inter Process Communication latency for the non real-time operating system may exceed 1ms as it is optimized to increase the average performance.

We built a distributed controller system composed of two controllers running a control application example given in Chapter 3. The experimental results have shown that, real-time packets of a control application can be transmitted within 6.6ms using D^3RIP with 3ms time-slots. It is possible to decrease latency for the tranmission of the real-time packets by decreasing the time-slot duration. The limiting factor in determining time-slot duration is the time synchronization performance and the Inter Process Communication latency for the real-time operating system.

In this thesis, time synchronization was implemented by using the timestamp of clock source included in the synchronization packets. However, as those packets traverse through the network they are exposed to network jitter which decreases our accuracy in measuring synchronization performance. As a possible solution to isolate the overheads caused by the network, it is possible in future work to measure the synchronization accuracy by connecting Pulse Per Second output pin of system clocks to an oscillator and measure the exact difference between clocks. It was not possible to build this configuration as the system clocks of our systems did not have such an output signal.

D^3RIP has superior bandwidth utilization compared to other protocols but it has some drawbacks too. First, it is required to employ a proper control mechanism to validate the correct operation of the protocol at the runtime as it relies on distributed computations. In addition, the configuration for the DART protocol is not easy for rather complex control applications.

REFERENCES

[1] A. K. Gözcü, *Implementation and Evaluation of a Synchronous Time-Slotted Medium Access Protocol for Networked Industrial Embedded Systems* MSc. Thesis, METU Sept. 2011

[2] www.rt.eei.uni-erlangen.de/FGdes/faudes/index.html (last accessed on 29/08/2011)

[3] http://ptpd.sourceforge.net/ (last accessed on 29/08/2011)

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

[5] J.R. Moyne, D.M.Tilbury, *Control and Communication Challenges in Networked Real-Time Systems*. Proceedings of the IEEE , vol.95, no.1, pp.29-47, Jan. 2007.

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

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

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

[9] J. C. Eidson, *Measurement, Control, and Communication Using IEEE 1588*. 2009.

[10] Kendall Correll, Nick Barendt Michael Branicky *Design Considerations for Software Only Implementations of the IEEE 1588 Precision Time Protocol* Proc. Conference on IEEE-1588 Standard for a Precision Clock Synchronization Protocol for Networked Measurement and Control Systems, NIST and IEEE, 2005

[11] 1st IFAC Workshop on Dependable Control of Discrete Event Systems, 2007

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

[13] M. Felser and T. Sauter, *Standardization of industrial Ethernet - The Next Battlefield?* IEEE International Workshop on Factory Communication Systems, 2004.

[14] M. Felser and T. Sauter, *Real-time ethernet the EtherCAT solution,* Comput. Control Eng. J., vol. 15, no. 1, pp. 1621, Feb.Mar. 2004.

[15] E. Schemm, *SERCOS to link with ethernet for its third generation,* Comput. Control Eng. J., vol. 15, no. 2, pp. 3033, Apr.May 2004

[16] *Real-Time Ethernet: PROFINET IO: Proposal for a Publicly Available Specification for Real-Time Ethernet,* Doc. IEC 65C/359/NP, 2004

[17] L. Liu and G. Frey, *Simulation approach for evaluating response times in networked automation systems,* Emerging Technologies & Factory Automation, 2007. ETFA. IEEE Conference on , vol., no., pp.1061-1068, 25-28 Sept. 2007

[18] K. Schmidt, E. Schmidt, and J. Zaddach, *Safe operation of distributed discrete-event controllers: A networked implementation with real-time guarantees,* IFAC World Congress, 2008.

[19] K. Schmidt, E. Schmidt, and J. Zaddach, *Distributed real-time protocols for industrial control systems: Framework and examples,* Submitted to IEEE Transactions on Parallel and Distributed Systems, 2011.

[20] Christos G. Cassandras and Stephane Lafortune, *Introduction to Discrete Event Systems,* Springer, 2nd edition, 2007.

[21] J. J. Labrosse, *uC/OS-III, The Real-Time Kernel, or a High Performance, Scalable, ROMable, Preemptive, Multitasking Kernel for Microprocessors, Microcontrollers & DSPs* 2009

[22] Shen, Wu, Li, Zhang, *Research of The Real-time Performance of Operating System*. 2009.

[23] Barr, Michael, *Choosing an RTOS, Embedded Systems Programming*. January 2003.

[24] http://www.osadl.org (last accessed on 29/08/2011)

[25] http://www.gnu.org/licenses/gpl.html (last accessed on 29/08/2011)

[26] Elsir, Sebastian, Voon, *A RTOS for Educational Purposes*

[27] www.kernel.org/pub/linux/kernel/v2.6/linux-2.6.33.7.tar.bz2 (last accessed on 29/08/2011)

[28] www.kernel.org/pub/linux/kernel/projects/rt/older/patch-2.6.33.7.2-rt30.bz2 (last accessed on 29/08/2011)

[29] http://www.ines.zhaw.ch/en/engineering/ines/ieee-1588/hardware.html (last accessed on 29/08/2011)

[30] H. Weibel, D. Bechaz, *IEEE 1588 Implementation and Performance of Time Stamping Techniques*. 2004.

[31] S. Johannessen, *Time synchronization in a local area network. Control Systems Magazine* April. 25-28 2004

[32] G.-S. Tian, Y.-C. Tian and C. Fidge, *High-Precision Relative Clock Synchronization Using Time Stamp Counters* Engineering of Complex Computer Systems, 2008. ICECCS 2008.

[33] http://www.national.com/pf/DP/DP83640.html (last accessed on 29/08/2011)

[34] http://www.uml.org (last accessed on 29/08/2011)